Efficient Object Sharing in Shared-Memory Multiprocessors

by

Mark Moir

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1996

Approved by:

Prof. James Anderson, Adviser

Prof. Maurice Herlihy, Reader

Prof. Don Stanat, Reader

Mark Moir. Efficient Object Sharing in Shared-Memory Multiprocessors
(Under the direction of Professor James H. Anderson.)

Abstract

The goal of this work is to facilitate efficient use of concurrent shared objects in asynchronous, shared-memory multiprocessors. Shared objects are usually implemented using locks in such systems. However, lock-based implementations result in substantial inefficiency in *multiprogrammed* systems, where processes are frequently subject to delays due to preemption. This inefficiency arises because processes can be delayed while holding a lock, thereby delaying other processes that require the lock.

In contrast, *lock-free* and *wait-free* implementations guarantee that the delay of one process will not delay another process. We show that *lock-free* and *wait-free* implementations for shared objects provide a viable alternative to lock-based ones, and that they can provide a significant performance advantage over lock-based implementations in multiprogrammed systems.

Lock-free and wait-free implementations are notoriously difficult to design and to verify as correct. *Universal constructions* alleviate this problem by generating lock-free and wait-free shared object implementations using sequential implementations. However, previous universal constructions either require significant creative effort on the part of the programmer for each object, or result in objects that have high space and time overhead due to excessive copying.

We present lock-free and wait-free universal constructions that achieve low space and time overhead for a wide spectrum of important objects, while not placing any extra burden on the object programmer. We also show that the space and time overhead of these

constructions can be further reduced by using k-exclusion algorithms to restrict access to the shared object. This approach requires a long-lived renaming algorithm that enables processes to acquire and release names repeatedly. We present several efficient k-exclusion and long-lived renaming algorithms.

Our universal constructions are based on the load-linked and store-conditional synchronization instructions. We give a constant-time, wait-free implementation of load-linked and store-conditional using compare-and-swap, and an implementation of multi-word load-linked and store-conditional instructions using similar one-word instructions. These results allow our algorithms and others to be applied more widely; they can also simplify the design of new algorithms.

Acknowledgements

First, I want to thank my adviser, Jim Anderson, for his support, enthusiasm, and patience. I have learned an enormous amount from working with him, and have thoroughly enjoyed doing so. I am also grateful to Jim for arranging financial support for me for several years, and to the alumni whose generous donations funded a fellowship for me this year.

Thanks, too, to the rest of my committee: Maurice Herlihy, Kevin Jeffay, Lars Nyland, Jan Prins, Don Stanat, and Steve Weiss. I greatly appreciate their willingness to bend their schedules to accommodate countless proposals, oral exams, defenses, proposals to propose, meetings to schedule proposals to propose, and so on. I am particularly grateful to Maurice Herlihy, who has endured much of this pain by means of phone and video conference, and has also been extremely supportive and encouraging of my work.

In addition, my work has benefited from lively discussions with many other colleagues over the past few years. The following list cannot possibly be complete, but it's a start: Yehuda Afek, Rajeev Alur, Hagit Attiya, Harry Buhrman, Rik Faith, Juan Garay, Steve Goddard, Jaap-Henk Hoepman, Michael Merritt, Gary Peterson, Srikanth Ramamurthy, Nir Shavit, Gadi Taubenfeld, Dan Touitou, Mark Tuttle, Paul Vitanyi, and Jae-Heon Yang. I would also like to acknowledge Phil McKinley and Chuck Severance of Michigan State University for their assistance with the use of their BBN GP1000 multiprocessor, Argonne National Laboratories for providing access to their Sequent Symmetry, and Lars Nyland for his help with the KSR performance studies in Section 5.4.

Life would be much more difficult (and no fun) without the support and companionship of many good friends. There is no chance of thanking them all personally here, but they know who they are, and they know that I owe them one. Thanks to all of them!

I have been very fortunate to have the love, support, and encouragement of a wonderful family. Despite being many thousands of miles away, they have never let me forget that they are there for me, and have always been extremely helpful to me. I am grateful to my parents for encouraging my education from a young age, and to my siblings for taunting me at every opportunity.

Finally, I am indebted to my wife Vikki. She has been incredibly supportive of me over the past few years, and has tolerated my long work hours with cheerful humor. I cannot thank her enough for being such a wonderful wife and friend. I am also grateful to her for writing this paragraph for me. (Just kidding!)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Shared objects provide a convenient means of communication and synchronization between concurrent processes in shared-memory multiprocessor applications. A *shared object* is a data structure (for example, a queue) that is shared among a collection of concurrent processes by means of a fixed set of *operations* (for example, enqueue and dequeue). This dissertation is concerned with the efficient implementation of shared objects. We assume a programming model in which a collection of asynchronous processes repeatedly perform some local computation (with which we are not concerned), and invoke operations on objects. Each operation has associated parameters and a return value. Our goal is to provide efficient implementations of these operations that appear to be executed atomically, giving the correct return value for the given parameters.

In order to avoid corruption of a shared object, it is usually necessary to synchronize concurrent accesses to the object. Various approaches have been proposed for dealing with the problem of coordinating concurrent accesses to a shared object. Among these approaches, the use of mutual exclusion [29] is the most commonly accepted. In order to modify a shared object using mutual exclusion, a process first acquires a lock associated with that object, performs its operation, and then releases the lock. Another process that needs to modify the object must wait until the lock is released.

1.1 Mutual Exclusion Algorithms

Substantial research effort has been devoted to the design of algorithms for mutual exclusion. Some researchers have focused on ensuring starvation freedom [60], so that a process that tries to acquire a lock eventually succeeds in doing so. Others have sought stronger guarantees that ensure, for example, that processes acquire the lock in (approximately) the same order that they request it [62]. Of course, significant effort has also been put into designing "efficient" mutual exclusion algorithms. Many have focused on the worst-case time complexity of acquiring a lock when no other processes require it, while others have studied the performance of various mutual exclusion algorithms under increasing levels of contention [12, 38, 72, 97]. (The *level of contention* is the number of processes that simultaneously request a lock.) Finally, efforts have been made to determine the impact of the available hardware and instructions on the design of mutual exclusion algorithms.

This effort has resulted in several mutual exclusion algorithms that perform very well and are widely used in shared-memory multiprocessor applications. However, all of these algorithms share a disadvantage that is fundamental to the use of mutual exclusion: by definition, only one process can access a given shared object at a time. This has several implications.

First, while one process holds a lock, any other process requiring that lock must wait. Waiting can be in the form of *busy waiting* or *blocking*. In the case of busy waiting, a process repeatedly tests a condition, for example by reading a shared variable, until the process determines that it can proceed. In the case of blocking, the process relinquishes the processor, thereby allowing another process to run. Both forms of waiting have their disadvantages. If a process busy waits for a long time, then no useful work is achieved on that processor during that time. On the other hand, if a process blocks, then its context must be saved and the context of another process restored. If the waiting time is short, then the expense of these two context switches may exceed the expense of busy waiting. Because of the difficulty of accurately predicting waiting times, either approach can lead to poor performance.

Second, in modern multiprocessors, processes can be delayed for a variety of reasons, including page faults, cache misses, interrupt handling, and preemption. If process delays are frequent, then the time spent waiting for locks can severely impact the performance of the application. This is particularly problematic in multiprogrammed applications, where processes are frequently subject to relatively long delays due to preemption. In this case, if a process is preempted while holding a lock, all other processes that require that lock must wait. The difficulty of efficient synchronization in multiprogrammed environments often leads programmers to avoid multiprogramming altogether. This prevents them from using the number of processes naturally dictated by their applications, and therefore significantly complicates their implementations. We hope that our work will help to alleviate this problem. With this goal in mind, we have attempted to make our algorithms as widely applicable as possible by avoiding the use of nonstandard operating system support for object sharing. In particular, this precludes the use of special schedulers that avoid the problems arising from concurrent object accesses by ensuring that they do not occur.

A third disadvantage of using mutual exclusion to implement shared objects is that it restricts parallelism: two operations on the same object cannot execute in parallel, even if they access disjoint parts of the object. This can severely limit scalability in multiprocessor applications if it causes processes to spend an excessive amount of time waiting for locks in order to perform operations on shared objects. In some cases this problem can be addressed by using finer locking granularity. In other words, by employing several locks for one object, operations that access disjoint parts of the object can execute in parallel. Unfortunately, this approach requires static information about the behavior of the operations, and often requires significant creativity on the part of the object programmer.

Finally, in priority-based systems, the use of mutual exclusion can cause priority inversion. A *priority inversion* arises when a process is preempted while holding a lock, thereby causing a higher-priority process that requires the same object to wait. This phenomenon is particularly problematic in hard real-time systems (which are often designed using priority-based schedulers). In such systems, processes (or tasks) are required to complete execution by a specified *deadline*; priority inversion can introduce lengthy delays, thereby causing tasks to miss their deadlines. Common solutions to this problem (for example, the priority inheritance protocol [83]) entail additional overhead and complicated operating system support.

For all of the reasons discussed above, it is highly desirable to reduce or eliminate waiting in concurrent applications. A major component of the research presented in this dissertation focuses on shared object implementations that avoid the use of locks and thereby greatly reduce, or even eliminate, waiting.

Previous efforts to eliminate waiting from shared object implementations have focused on lock-free and wait-free shared object implementations [44, 66]. A shared object implementation is *lock-free* if, for every operation by each process p, *some* operation is guaranteed to complete after a finite number of steps of process p. An implementation is *wait-free* if, for every operation by each process p, *that* operation is guaranteed to complete after a finite number of steps of process p. Note that both requirements preclude the use of locking: if some process is delayed for a long time while it has an object locked, then *no* process is able to complete an operation on that object.

The definition of a wait-free or lock-free implementation implies that multiple object operations can be performed concurrently (because if one is delayed, another must still be able to complete). Thus, the correctness condition for such implementations is necessarily more complicated than for mutual-exclusion-based implementations (where the correctness of the sequential operations implies the correctness of the implementation). Like most researchers in the area of wait-free and lock-free shared object implementations, we use *linearizability* [48] as a correctness condition for our constructions. As described more precisely in the next chapter, linearizability requires that processes invoking operations on the implemented object cannot distinguish between the wait-free or lock-free implementation of that object, and one that ensures that each operation is executed sequentially.

Lock-free and/or wait-free implementations have been developed for various shared objects. Unfortunately, lock-free and wait-free algorithms are notoriously difficult to design and to verify as correct. As a result, the design of these implementations, even for simple objects such as queues, has required significant creative and intellectual effort. To allow programmers of concurrent applications to easily use new lock-free and wait-free shared objects, a mechanism that *automatically* generates such an implementation from its sequential implementation is desirable. Towards this end, Herlihy proposed universal constructions [41]. A *universal construction* is a mechanism that produces a lock-free or wait-free implementation of a shared object, given code for a sequential implementation of that object.

In a seminal paper, Herlihy presented the first attempt at practical lock-free and wait-free universal constructions [42, 44]. While this line of research provides an excellent starting point for our work, this and other universal constructions [56, 77] for lock-free and wait-free shared objects have entailed significant time and space overhead, precluding their use in practical settings.

The main thesis to be supported by the work in this dissertation is that

> *lock-free and wait-free implementations for shared objects provide a viable alternative to lock-based ones, and that they have a significant advantage over lock-based implementations if process delays are common, as is the case in multiprogrammed systems.*

Many of the results presented in this dissertation support this thesis directly, while others are useful by-products of this research. Specifically, we present several new lock-free and wait-free constructions that substantially improve on the time and space overhead associated with similar previous constructions.

We also present several results involving synchronization instructions, such as load-linked and store-conditional (our universal constructions and previous ones are based on these instructions), including a constant-time, wait-free implementation of the load-linked and store-conditional instructions using compare-and-swap, and an implementation of multi-word load-linked and store-conditional instructions using similar one-word instructions. These results allow our algorithms and others to be applied with greater flexibility; they can also simplify the design of new algorithms. Finally, we study the *long-lived* renaming problem. This problem is important in algorithms — including some of ours — that have time complexity that depends on the size of the range of process identifiers. A fast, long-lived renaming algorithm can be used to reduce the size of this range, thereby improving the performance of the algorithm.

The remainder of this introduction is devoted to a brief discussion of all of this work. We begin with a brief description of Herlihy's universal constructions, which provide an important foundation for much of the work presented here. Other related work is discussed in Chapter 2.

1.2 Herlihy's Constructions

Herlihy presents three universal constructions: lock-free and wait-free constructions for "small" objects, and a lock-free implementation for "large" ones. A small object is one that can be copied in its entirety without excessive overhead. In Herlihy's lock-free implementation for small objects, a process p performs each operation on a *copy* of the object, rather than on the current version of the object. This copy is "owned" exclusively by process p; the operation can therefore be performed using purely sequential code, safe from interference by other processes. Having performed its operation on a local copy, process

p then attempts to make that copy "current" by modifying a shared pointer; the shared pointer always points to a copy that contains the current value of the implemented object. There is a risk that the effects of an operation by process p might be lost because another process q modifies the shared pointer immediately after p does. This would lead to an incorrect implementation, because q's operation would be applied to an object value that did not include p's changes to the object. To allow process p to detect this interference, and to retry its operation, the shared pointer is manipulated by two special instructions: load-linked (LL) and store-conditional (SC). These operations are used as a pair: a process p first reads the pointer using LL, and later attempts to write the pointer using SC. The SC of p *fails*, returning *false*, if a successful SC operation is executed by another process between the LL and SC of p. Thus, in the scenario described above, process q's SC fails (has no effect on the shared pointer), and q retries its operation from the beginning. But note that a SC fails only if another process successfully modifies the shared pointer, thereby completing an operation. Therefore, if a process q repeatedly fails to perform its operation, then some other process repeatedly completes an operation, so this implementation is lock-free. However, q could potentially fail forever because another process might interfere every time q attempts to modify the shared pointer. Therefore, the implementation is not wait-free.

Herlihy's wait-free implementation for small objects overcomes this problem by the addition of a "helping" mechanism, which ensures that if some process p repeatedly interferes with q's operation then p (or some other process) will eventually "help" q by performing q's operation along with its own. This helping is achieved by having each process that performs an operation first "announce" the operation by recording the operation to be performed, the parameters to be used, and some other administrative information. Then, when a process performs its operation on a local object copy, it checks for outstanding operations that have been announced by other processes, and also performs these operations on the local copy before attempting to update the shared pointer (using SC). Herlihy shows that this ensures that a process can fail to perform its own operation at most twice before the operation is performed by another process [44]. Several subtle difficulties arise from the need to ensure that each operation takes effect on the object exactly once, and that, when an operation *is* performed by a process other than the process that invoked it, the correct return value for that operation is communicated to the invoking process. Ensuring that these requirements are met is complicated by the possibility of a process p successfully performing an operation of another process q and subsequently being delayed indefinitely; this delay must not prevent process q from determining that its operation has been completed, or from ascertaining the return value of that operation. These problems are overcome by recording the return value of the operation, as well as the fact that the operation has been successfully performed, in the same atomic step in which the operation takes effect (that is, when the successful SC is performed). To facilitate this "multi-purpose" atomic step, return values and other administrative information are kept with each object copy.

Finally, Herlihy's lock-free implementation for large objects allows the object to be partitioned into blocks that are linked together by pointers. Now, each process that performs an operation maintains a *logically* distinct version of the object, although some parts of the object that are not affected by the operation may be *physically* shared by two or more logical copies. The goal of this approach is to avoid copying as much of the object as possible by allowing the "new" object value to share a substantial portion of the object with the "current" object value.

Each of Herlihy's implementations has its disadvantages. First, the guarantee of progress provided by his wait-free implementation for small objects comes at a cost: each time process p performs an operation, p must check each other process to see if it needs help. Thus, if N — the number of processes — is large, then there is a significant overhead involved in performing an operation, even if no other process concurrently accesses the object. Furthermore, in both the lock-free and wait-free implementations for small objects, each operation copies the entire object for each operation. (In fact, in the event of failed SC instructions, these operations must copy the object more than once in order to retry.) Depending on the size of the object, this can have a significant impact on performance. Finally, each process requires a local copy of the object. Again, if N is large, or of the object itself is large, this could represent a significant storage overhead.

The approach taken in Herlihy's lock-free implementation for large objects requires the object designer to explicitly partition the object into blocks, and to determine how these blocks should be updated and copied. This makes the implementation much more difficult to use. Also, for many common objects, this approach provides no advantage over the small-object implementations because the whole object must be copied anyway. For example, when implementing a FIFO queue as a linked list of blocks, the enqueue operation must copy the entire object because, in order to link in a new block, the "next" pointer in the last block must be changed, which necessitates copying the last block, which in turn necessitates modifying the next-to-last block, and so on. This "cascading" effect causes Herlihy's construction to copy the entire object in this case. Finally, Herlihy does not present a wait-free implementation for large objects.

1.3 Our Contributions

In Chapters 5 and 6, we present new techniques for implementing lock-free and wait-free shared objects.[1] These new techniques are designed to overcome all of the problems described above. Specifically, in Chapter 5, we present new lock-free and wait-free universal constructions for large objects, and, in Chapter 6 we present a technique that allows the time and space overhead of universal constructions to be tied to *expected*, rather than *worst case*, contention.

1.3.1 Universal Constructions for Large Objects

Our large-object constructions provide the programmer with the "illusion" of a large, contiguous array within which to implement a shared object. This is a natural programming paradigm for many applications as it essentially models physical memory. In reality, the array is implemented by a collection of smaller blocks. When an operation modifies a part of this array, a new block replaces the block containing that part. The address translation and record-keeping necessary to provide the illusion of a contiguous array is performed entirely by our constructions, and is transparent to the object programmer.[2]

[1] As explained in Chapter 6, the techniques presented there are not, technically speaking, wait-free. However, they are designed to allow implementations to be "tuned" for a particular application so that they provide the same benefits that wait-free implementations provide, while keeping overhead low.

[2] To use our constructions, a programmer must supply sequential code for operations and also determine the size of the blocks that implement the underlying array. Therefore, our constructions are not com-

Thus, unlike Herlihy's large-object construction, ours shield the programmer from managing the details of concurrent object accesses. Moreover, because the programmer now deals with logical addresses (offsets into the imaginary array), rather than physical addresses, the replacement of a block does not necessitate modifications to other blocks (as it does in Herlihy's large object construction). Thus, the cascading effect described earlier does not occur in implementations that use our constructions. As a result, our constructions perform significantly better than Herlihy's for some objects, while having much lower space overhead.

1.3.2 Using k-Exclusion to Further Reduce Overhead

The techniques presented in Chapter 6 are motivated by the observation that, in most applications, it is unlikely that all N processes in the system would access the same object concurrently, and that even if this *did* happen, it would be very unlikely for $N-1$ of them to be delayed simultaneously. Thus, the requirement of wait-freedom — that a process is guaranteed to complete its operation even if the other $N-1$ processes are delayed — is stronger than most applications require. The key idea behind the results in Chapter 6 is to provide weaker progress guarantees than wait-freedom, while still providing *some* resilience to process delays, and to do so with less overhead than wait-free constructions. This is achieved by using a wait-free object implementation for $k < N$ processes. Now, time and space complexity are dependent on k, rather than on N, and, provided at most k processes concurrently access this implementation, none will ever have to wait to complete

pletely transparent. Nonetheless, the choice of block size affects only performance; correctness is guaranteed regardless of the block size chosen.

its operation. Unfortunately, most wait-free implementations for k processes will behave *incorrectly* if accessed simultaneously by more than k processes. Therefore, it is important to *ensure* that the assumed bound of k processes accessing the k-process, wait-free object implementation is never violated. This is achieved by the use of a k-exclusion algorithm. The *k-exclusion problem* is a generalization of the mutual exclusion problem, in which up to k processes may execute within their critical sections concurrently. In Chapter 6, we present several fast k-exclusion algorithms for a variety of architectures, and present performance studies that show that this approach can result in objects that perform better under multiprogramming than objects implemented using either Herlihy's wait-free construction for small objects, or a state-of-the-art mutual exclusion algorithm. These techniques also reduce the space overhead of lock-free constructions, but are lock-free if contention does not exceed k.

1.3.3 Fast, Long-Lived Renaming

In order to use most k-process object implementations, a process needs a unique identifier from $\{0..k-1\}$. In the k-exclusion-based approach described above, different sets of processes use the k-process object implementation at different times, so these identifiers cannot be assigned statically. Therefore, a mechanism is required whereby processes can repeatedly acquire and release identifiers (otherwise known as names). This can be achieved through the use of a fast, long-lived renaming algorithm. In the *long-lived renaming problem*, processes with distinct names ranging over 0 to $N-1$ repeatedly acquire and release names from $\{0, ..., M-1\}$, for some $M < N$. It is assumed that at most k processes request or hold names concurrently. A long-lived renaming algorithm is called *fast* if the time complexity of acquiring and releasing a name once is independent of N and polynomial in k. In Chapter 7, we present efficient long-lived renaming algorithms that employ "strong" synchronization instructions such as test-and-set and fetch-and-add, among other, less well-known instructions. With the hope of finding more portable renaming algorithms, we have also studied fast, long-lived renaming algorithms that employ only simple read and write instructions. The restriction to reads and writes makes the problem of renaming much more difficult, and in fact renders impossible the goal of renaming to a name space smaller than $2k-1$. We present the first fast, long-lived renaming algorithm that uses only reads and writes and renames to the optimal name space size of $2k-1$. We also present the first fast, one-time renaming algorithm, which achieves a name space of size $k(k+1)/2$. (One-time renaming is a special case of long-lived renaming in which processes do not release names.)

1.3.4 Support for Wider Applicability

Our constructions as well as Herlihy's are based on LL and SC instructions. As discussed in the next chapter, Herlihy has shown that, in order to implement *some* objects in a wait-free (or lock-free) manner, "strong" synchronization instructions such as LL/SC or compare-and-swap (CAS) are *necessary*. Thus, because universal constructions can be used to implement *any* object, they are necessarily based on such instructions. In Chapter 4, we present several results that allow our constructions (and other LL/SC-based algorithms) to be applied in a wider variety of settings, including those that provide CAS and not LL/SC, and some that, for practical reasons, do not provide LL/SC instructions with the semantics we assume.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Related work is discussed in Chapter 2. Notational conventions and definitions are listed in Chapter 3. In Chapter 4, we present several mechanisms that allow the algorithms presented in Chapters 5 and 6 to be applied with greater flexibility. Chapter 5 contains lock-free and wait-free constructions for large shared objects. In Chapter 6, we present mechanisms for reducing the space overhead and improving the performance of lock-free and wait-free shared object implementations. In Chapter 7, we study the renaming problem. Finally, conclusions and a discussion of future directions for this research appear in Chapter 8. Some of the lengthier proofs from Chapters 5 through 7 appear in appendices. (This dissertation includes work that is based, with permission, on previously published work [8, 9, 10, 74, 75].)

Chapter 2

Related Work

2.1 Mutual Exclusion Algorithms

In the *mutual exclusion*, each of a set of concurrent processes repeatedly executes a *noncritical section* of code and a *critical section* of code. A solution to the mutual exclusion problem consists of code for an *entry section* to be executed before the critical section, and for an *exit section* to be executed after critical section. These code sections must ensure that no two distinct processes execute within their critical sections concurrently. This problem was first posed by Dijkstra in 1965 [29], and an enormous amount of research effort has been devoted to studying this problem since. In this section, we give a brief overview of the recent history of research on mutual exclusion algorithms for shared-memory multiprocessors.

The simplest mutual exclusion algorithm is known as a *test-and-set* lock. This algorithm uses a single bit, which contains zero when the lock is free, and one when some process is holding the lock. In order to acquire the lock, a process repeatedly attempts to change the bit from zero to one using the test-and-set instruction (which is defined in

Chapter 3). While another process holds the lock, these attempts fail, so processes that are requesting the lock must retry.

Recent performance studies [12, 38, 72, 97] show that, as the number of concurrent processes in a system grows, excessive traffic on the processor-to-memory interconnect can quickly become a bottleneck that limits performance. Therefore, in order to achieve "scalable" performance (that is, performance that does not degrade unacceptably as the number of processes in the system grows), it is desirable to design mutual exclusion algorithms that minimize the amount of interconnect traffic generated by each process. Furthermore, "strong" synchronization instructions such as test-and-set, which both read and write memory in one atomic step, generally take longer to execute than instructions that simply read or write.

Given these observations, the test-and-set lock can be improved by having each process repeatedly read the lock bit until it is zero, and then attempt to set it to one using test-and-set. The resulting algorithm is known as a *test-and-test-and-set* lock. This algorithm reduces the number of expensive test-and-set instructions executed and, in machines with coherent cache mechanisms, allows processes to avoid generating unnecessary interconnect traffic while the lock is held by another process. However, two disadvantages remain. First, when the lock is released, a "flurry" of interconnect activity can arise as each waiting process attempts to acquire the lock. Also, neither the test-and-set lock, nor the test-and-test-and-set lock provide any progress guarantees: it is possible for one process to repeatedly fail to acquire the lock, resulting in starvation.

Recently, various researchers have presented mutual exclusion algorithms for shared-memory multiprocessors that are designed to be scalable (by avoiding excessive interconnect traffic) while providing progress guarantees to processes that attempt to acquire the lock. Some of the best-known are due to Anderson [12], to Graunke and Thakkar [38], to Mellor-Crummey and Scott [72], and to Yang and Anderson [97]. All of these mutual exclusion algorithms achieve scalable performance through the use of "local spinning".

A *spinning* algorithm is one that busy-waits by repeatedly testing shared variables (i.e., no shared variables are written while busy-waiting). A *local spinning* algorithm is one in which all busy-wait loops access only shared variables that are *locally-accessible*, i.e. that do not require a traversal of the interconnect. Two classes of shared-memory multiprocessors — distributed shared-memory multiprocessors and cache-coherent multiprocessors — lend themselves to the use of local spinning. On a distributed shared-memory machine, a shared variable is locally-accessible if it is stored in a local partition of shared memory. On a cache-coherent machine, a shared variable is locally-accessible if a copy of that variable currently resides in the local cache.

To see the impact of local spinning on the performance of various mutual exclusion algorithms, consider Figure 2.1. This figure summarizes results of performance experiments run on the Sequent Symmetry, a shared-memory multiprocessor with cache-coherence; these results were originally reported by Yang and Anderson in [97]. The graph compares a simple test-and-set lock with mutual exclusion algorithms by Peterson and Fischer [82], by Lamport [65], by Styer [87], by Yang and Anderson [97], by Mellor-Crummey and Scott [72], and by Anderson [12]. Each point (x, y) in the graph represents the average time y for one critical section execution with x competing processes, one per processor, averaged over 100,000

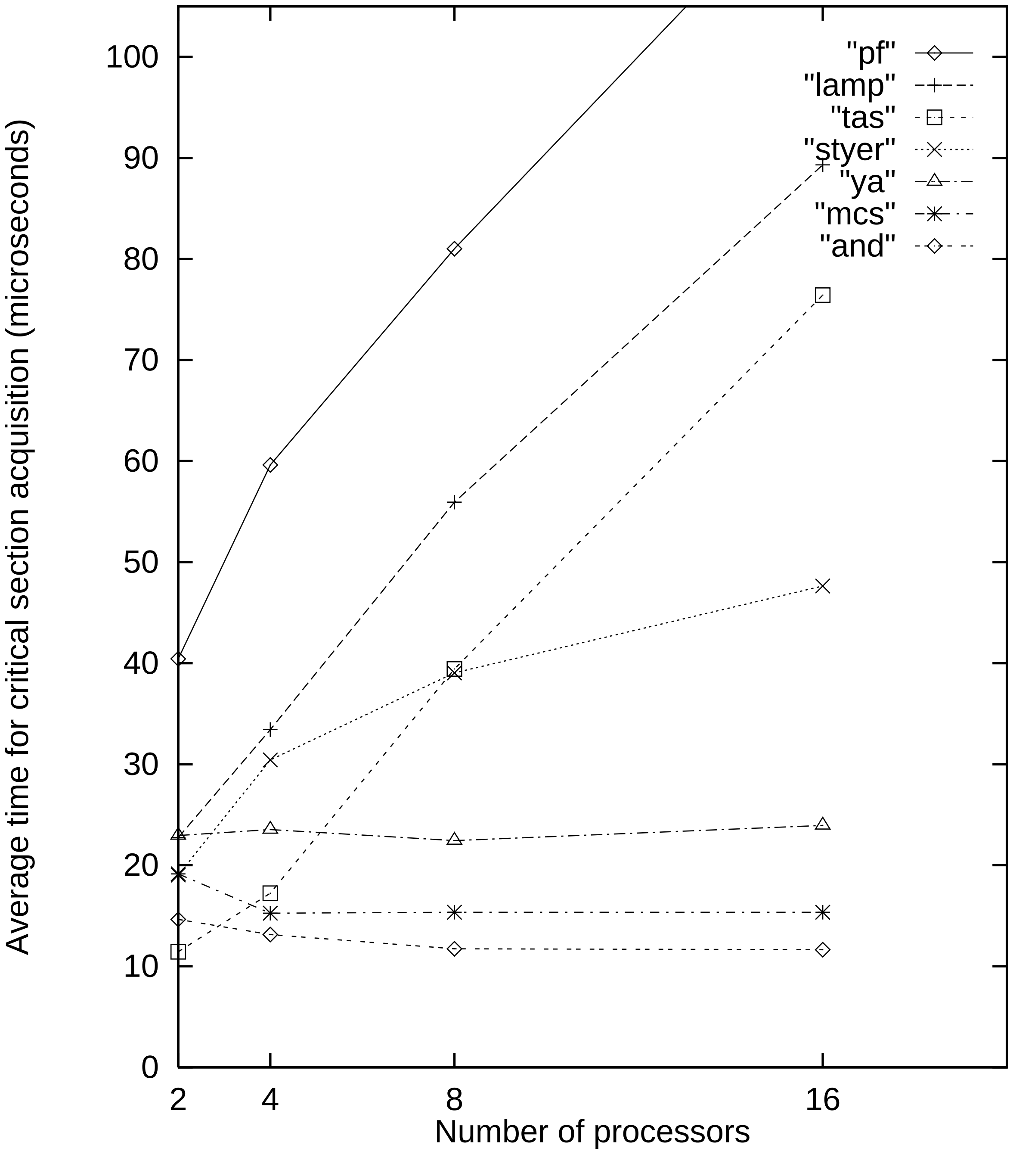


Figure 2.1: Performance comparison of a test-and-set lock (tas), with mutual exclusion algorithms by Peterson and Fischer (pf), Lamport (lamp), Styer (styer), Yang and Anderson (ya), Mellor-Crummey and Scott (mcs), and Anderson (and).

critical section executions. The algorithms by Anderson, by Mellor-Crummey and Scott, and by Yang and Anderson are the only algorithms tested in which processes locally spin. Each of these local-spin algorithms yields a relatively flat curve in Figure 2.1, indicating good scalability. The curves for all of the other algorithms rise more quickly, indicating poor scalability.

Despite these efforts to provide scalable mutual exclusion algorithms for a variety of architectures providing a variety of machine instructions, a serious problem remains. If a process is delayed while holding a lock for a shared object, then no other process can access that object until that process resumes execution and releases the lock. As a result, other processes might spin for a relatively long time waiting for the lock, and achieve no useful work in that time.[1] If processes are multiprogrammed, this problem is exacerbated by the fact that, in most of the mutual exclusion algorithms mentioned above, processes acquire locks in a FIFO order. In this case, if the process that is "next in line" to acquire a lock is preempted while waiting for the lock, then all other processes must wait for that process to be rescheduled before it even acquires the lock. Those processes might in turn be preempted while they are waiting, making matters even worse. This problem seems to be inherent in any FIFO mutual exclusion algorithm.

Recently, Wisniewski *et al.* [96] have designed mutual exclusion algorithms with the relaxed condition that *running* processes acquire the lock in FIFO order. This avoids the problem described above, in which processes wait in line behind preempted processes.

[1] Another alternative approach, called *idle waiting*, is to allow a waiting process to relinquish the processor to another process. This approach assumes that there is always another process that can perform useful work. Furthermore, a non-trivial overhead is usually incurred when switching from one process (or thread) to another. Thus, the use of idle-waiting can lead to poor performance if waiting times are usually short. For a detailed description of these trade-offs, and for a study of heuristics for deciding between idle waiting and busy waiting, see [69].

However, as a result of this relaxed condition, these algorithms admit the possibility of starvation. Furthermore, implementing the relaxed FIFO ordering and ensuring that a process is not preempted while in its critical section requires the use of special kernel support. Clearly, shared object implementations that do not rely on special kernel support are more widely applicable. In the next section, we describe recent research that has attempted to circumvent this problem by avoiding locking altogether, while being implemented without non-standard kernel support.

2.2 Lock-Free and Wait-Free Object Implementations

Lock-free and wait-free shared object implementations provide alternatives to mutual exclusion. A shared object implementation is *wait-free* if and only if each operation by each process p is guaranteed to complete after a finite number of steps by that process. To be *lock-free*, an implementation only needs to guarantee that an operation by *some* process (not necessarily p) will complete after a finite number of steps by process p. An implementation based on critical sections is neither lock-free nor wait-free because, while a process is delayed in its critical section, *no* process completes an operation.

In the remainder of this section, we introduce linearizability — the correctness condition used to verify lock-free and wait-free implementations, and describe previous research on such implementations. Specifically, we discuss the consensus problem, which provides fundamental insight into the hardware support required for these implementations, and also summarize previous work on lock-free and wait-free shared objects, both universal constructions and implementations of specific objects.

2.2.1 Linearizability

Implementations of lock-free and wait-free operations consist of code fragments that typically execute multiple atomic statements. As a result, each operation invocation occurs in an "interval" of time. Because processes may invoke operations concurrently with each other, two or more of these intervals may overlap. This gives rise to a partial order over invocations: if one invocation completes before another invocation begins, then the first invocation precedes the second in the partial order; if two invocations overlap, then they are not ordered. The semantics of a shared object is satisfied if each invocation on an object appears to the invoking processes to be executed instantaneously at some distinct point during the invocation's interval. The formal correctness condition used to ensure this is *linearizability* [48]. Linearizability requires that the partial order that arises from any series of invocations on an object can be extended to a total order in such a way that the values returned by the invocations in the total order are consistent with the sequential semantics of the implemented object.

2.2.2 The Consensus Hierarchy

In the *consensus problem*, a set of N asynchronous processes, each with a given *input value*, communicate in order to agree on the input value of one of the processes. Loui and Abu-Amara [70] showed that the consensus problem cannot be solved in a wait-free manner for $N > 1$ in a shared-memory multiprocessor that provides only simple read and write instructions. (A similar result was previously proved for message passing systems by Fischer, Lynch, and Patterson [36].)

Herlihy later extended these results to other shared objects by classifying each object according to its consensus number [41, 43]. The *consensus number* of an object is the maximum number of processes for which a wait-free consensus algorithm exists that relies only on that object and read and write instructions. Herlihy showed that for each $N \geq 1$, there exists an object with consensus number N. More importantly, he also showed that an object with consensus number N is universal in a system of N processes. An object is *universal in a system of N processes* if it can be used to implement *any* object in a wait-free manner in that system. These results give rise to a "hierarchy" of objects: each object is placed at the level of its consensus number and, in a system of N processes, it is impossible to construct a wait-free implementation of a shared object at level N using any object from a lower level. Herlihy also identified objects with *unbounded* consensus number — objects that can solve consensus in a system consisting of any number of processes. These results imply that objects with unbounded consensus number can be used to construct wait-free implementations of *any* object in a system of *any* number of processes. A well-known example of an object with unbounded consensus number is a register that supports read and compare-and-swap operations. This implies that, on a shared-memory multiprocessor that supports instructions (like compare-and-swap) with unbounded consensus number, any object can be implemented in a wait-free manner for any number of processes.

2.2.3 Universal Constructions

In recent years, several groups of researchers have presented methods for automatically "transforming" sequential object implementations into wait-free or lock-free ones [2, 18, 41, 43, 44, 56, 77, 84]. These methods are called *universal constructions*. A universal construction relieves the object designer of the need to reason about concurrency, thereby greatly simplifying the task of providing a correct lock-free or wait-free implementation for a particular shared object. Much of the research presented in this dissertation is based on universal constructions due to Herlihy [44]. Herlihy's lock-free and wait-free constructions for small objects, as well as his lock-free construction for large objects, are described in Section 1.2; in the following paragraphs, we briefly describe other universal constructions.

The first universal construction is due to Herlihy [41]. This construction is used to prove the universality of consensus, and is therefore based on general consensus objects. The construction works by appending operations onto a list in an order that is consistent with the partial order over operation invocations. The value of the object can then be deduced by "replaying" the operations applied since a known state. The consensus objects are used by processes to "agree" on the order in which operations are appended to the list. This construction requires an unbounded number of variables; Herlihy later presents a version that uses a bounded number of variables, although the values of some of these variables grow, albeit slowly, without bound [43]. In a subsequent paper, Jayanti and Toueg correct some minor errors and show that bounded variables suffice [56]. Despite all of these improvements, these constructions all have high time and space complexity, and cannot be considered practical.

Plotkin [77] proposed a new universal synchronization primitive called a "sticky bit", and presents a universal construction that is based on this primitive. A *sticky bit* is a variable that contains one of three values: 0,1, or $\perp$, and supports a read operation and a $Jam(v)$ operation (where $v \in \{0, 1\}$), which sets the variable to v and returns true if the variable contains v or $\perp$, and returns false otherwise. (A sticky bit also supports a "flush" operation, which sets the variable back to $\perp$ provided it is not executed concurrently with any other operation on the same sticky bit, in which case it has unpredictable results.) Plotkin shows that sticky bits can be used to implement a sticky byte — a multi-valued version of the sticky bit — and uses this in turn to provide a universal construction. This construction is similar in spirit to Herlihy's original construction [41], although it uses bounded space. It also has overhead that is too high for the construction to be considered practical.

Later, Herlihy [42] made an important step towards practical universal constructions by presenting the first universal construction that is based on a "real" machine instruction, namely compare-and-swap. Although implementable on existing shared-memory multiprocessors, this construction entails high overhead, and is therefore impractical.

Herlihy made the first serious attempt at practical universal constructions in [44]. He presented three constructions: lock-free and wait-free constructions for "small" shared objects, and a lock-free construction for "large" shared objects. These constructions are based on the LL/SC instruction pair, and are described in more detail in Section 1.2.

Although Herlihy's best constructions represent important progress towards practical universal constructions, they have several disadvantages. First, operations implemented using the wait-free construction have time complexity that is linear in the total number of processes, even if the operation accesses the object alone (that is, no other process accesses the object concurrently with that operation). This is because each operation must check, for every process, whether that process has an outstanding operation that needs to be helped.

Second, the small-object constructions copy the entire object. Herlihy's solution to this problem (his lock-free, large-object construction) requires some creativity on the part of the object programmer, fails to reduce copying for some common objects, and does not provide wait-free implementations. Finally, Herlihy's constructions do not allow operations to execute in parallel on the same object; this limits throughput to being no better than that of a sequential implementation. Several groups of researchers have presented constructions that are designed to overcome one or more of these disadvantages. These efforts are described briefly below.

Barnes [18] recognized the importance of allowing operations to execute in parallel where possible. He presented a mechanism in which an object is protected by a number of locks. In order to perform an operation on the object using Barnes's method, a process acquires the locks associated with all parts of the object that it accesses, performs its operation, and then releases the locks. Barnes uses a technique he calls the *cooperative technique* to avoid the problems that arise from the use of mutual exclusion, and to guarantee lock-freedom. With the cooperative technique, when a process acquires a lock, it attaches to that lock a pointer to a variable that describes the operation to be performed. This allows a process that needs to acquire a lock that is already held by another process to "help" the latter process to perform its operation and to release the lock. (A similar approach is proposed by Turek, Shasha, and Prakash [89].) Barnes's construction uses the LL/SC instruction pair to ensure that each step of an operation is performed only once, despite the possibility of multiple processes attempting to perform that step concurrently. Because of the relative expense of executing these synchronization instructions, this technique introduces significant overhead. The cooperative technique can lead to livelock if the implemented operations without the cooperative technique would lead to deadlock. Thus, as with traditional lock-based implementations, it is important to avoid this situation, and Barnes uses the common approach of ensuring that all processes acquire locks in the same order. This can be problematic for some operations, because it is difficult or impossible to determine in advance which locks the operation will acquire. Barnes uses a technique he calls the *caching method* to avoid this problem. Using the caching method, a process performs an operation on a private "copy" of the object (only the parts of the object affected by the sequential operation are copied), and then acquires the necessary locks in order, and, in the absence of any interference, performs its operation on the object using the cooperative technique. This technique avoids the livelock problem, but also introduces further overhead. Nonetheless, Barnes's construction improves over Herlihy's by avoiding copying the entire object, and by allowing operations to execute in parallel where possible. Barnes did not present a wait-free construction.

Shavit and Touitou presented a method that they called *software transactional memory* [84], which is based on *transactional memory* — a hardware mechanism proposed by Herlihy and Moss [45]. Transactional memory allows programmers to write transactions (code fragments that operate on shared memory) that either execute atomically or fail without modifying the memory. Shavit and Touitou's approach allows transactions to be executed in a lock-free manner, and also allows multiple transactions to execute in parallel, provided that that they do not interfere with each other. This is achieved through the use of a technique that is similar to Barnes's cooperative technique: processes acquire locks associated with the memory locations to be updated, and a location that is locked by another process can be released by "helping" that process. As in Barnes's implementation, the LL/SC instruction pair is used to ensure that a slow process does not corrupt the implemented object. A key difference between the two approaches is that software transactional memory avoids the situation in which a process recursively helps a long "chain" of overlapping operations, only to later fail as a result of the operations it helped. According to simulated performance studies conducted by Shavit and Touitou, this results in improved performance over Barnes's method. One limitation of software transactional memory (as presented in [84]) is that it does not support dynamic transactions. (A transaction is *static* if the locations it accesses are known in advance, and *dynamic* otherwise.) As a result, it is not particularly well suited to implementing general shared object operations. However, most common synchronization primitives — for which software transactional memory is intended — access a fixed set of locations, and can therefore be easily implemented using static transactions. Shavit and Touitou did not present a construction for wait-free transactions.

The constructions of Barnes and of Shavit and Touitou are lock-free, but not wait-free. Thus, they admit the possibility that a process will fail to complete its operation indefinitely. Herlihy's wait-free construction guarantees that this cannot happen, but does so at considerable expense. In particular, the time complexity of performing an operation is $\Omega(N)$, where N is the total number of processes, even if no other process attempts to perform an operation concurrently. Afek, Dauber, and Touitou [2] presented a wait-free, universal construction that attempts to achieve a middle ground: the time complexity of their construction depends on the number of processes that concurrently *access* the object, rather than the *total* number of processes. Thus, the performance of objects implemented using their constructions is determined by *actual* levels of contention, rather than *worst case*. In particular, if one process accesses the object alone, then performance should be comparable to that of a lock-free implementation. As in Herlihy's wait-free construction, progress is guaranteed by having processes "help" each other to perform operations. However, in Herlihy's construction, the helping mechanism requires processes to check each other process for an outstanding operation. Afek, Dauber, and Touitou's construction avoids this overhead by maintaining a list of processes that are *currently* accessing the object. Maintaining and checking this list is done with time complexity that depends on the size of the list, and not on the total number of processes. Afek, Dauber, and Touitou present several variations on this theme, including one that does not require the entire object to be copied by each operation. These constructions entail some ingenious new techniques, and represent important progress in the development of practical universal constructions. Specifically, they employ a mechanism that allows a list of processes that need help to be maintained in a wait-free manner. This list is accessed with time complexity that is linear in the number of processes in the list (not the total number of processes). The use of this technique allows these constructions to improve on the $\Omega(N)$ best-case time complexity of Herlihy's wait-free construction. However, these new constructions are not expected to perform well in practice, because they make heavy use of the LL/SC synchronization instructions. The LL/SC instructions are usually quite expensive, relative to simple read and write instructions [61]. Finally, these constructions do not exploit parallelism.

2.2.4 Specific Objects

Finally, many researchers have presented lock-free and wait-free implementations for specific shared objects. Such implementations can potentially take advantage of the semantics of the object under consideration to improve performance. However, most implementations of specific objects have required considerable creative and intellectual effort, highlighting the need for universal constructions. Some specific object implementations are listed below.

Many researchers have studied implementations of various kinds of wait-free shared objects using only read/write registers. These implementations include constructions of complex registers from simpler registers [22, 25, 39, 40, 58, 59, 66, 68, 78, 79, 81, 85, 88, 93]; atomic snapshots that allow multiple variables to be read atomically, [1, 4, 5, 16, 17, 32, 50, 55], algorithms for maintaining timestamps [31, 33, 37, 52], and mechanisms for implementing any object whose operations satisfy certain algebraic requirements [6, 7, 13]. For example, a construction is given in [7] that implements any object such that, for each pair of operations on the object, either the two operations commute with each other, or one overwrites the other (i.e., the effects of executing both operations is the same as executing just one of them).

Other researchers have considered wait-free and lock-free implementations using instructions that are stronger than simple reads and writes. Implementations of various types of queues have been presented by Lamport [64], by Herlihy and Wing [47], by Israeli and Rappoport [53], by Wing and Gong [94, 95], and by Michael and Scott [73]. Anderson and Woll [11] and Lanin and Shasha [67] present implementations for various set operations.

Valois presents lock-free implementations for various data structures, including queues, lists, trees, and dictionaries [90, 91, 92]. Finally, Massalin and Pu have implemented an entire operating system using lock-free data structures such as lists, queues, and stacks [71].

Chapter 3

Preliminaries

In this chapter, we briefly describe our model of computation, and introduce some conventions, definitions, and notation that are common to the following chapters.

All of the results in this dissertation concern *asynchronous, shared-memory multiprocessors.* This description includes a wide variety of architectures that consist of two or more processors connected to each other and to memory modules by an interconnection network, such as a bus or crossbar. These processors are used to run sequential processes (or threads; the distinction is not important here) that communicate using shared memory. This communication is achieved by means of simple memory instructions such as read and write operations, as well as more complicated instructions in some cases, such as test-and-set, fetch-and-add, CAS, and the LL/SC instruction pair.

We now describe the common instructions used in this dissertation. Other, less common, instructions are defined in the chapters in which they are used. Figure 3.1 contains atomic code fragments that are equivalent to each of the instructions discussed below. We

$$
\begin{array}{lll}
\mathit{test\_and\_set}(X) & \equiv & \mathit{tmp}, X \ := \ X, \mathit{true};\ \mathbf{return}\ \neg \mathit{tmp} \\
\mathit{fetch\_and\_add}(X, v) & \equiv & \mathit{tmp} := X;\ X := X + v;\ \mathbf{return}\ \mathit{tmp} \\
\mathit{CAS}(X, v, w) & \equiv & \mathbf{if}\ X = v\ \mathbf{then}\ X := w;\ \mathbf{return}\ \mathit{true}\ \mathbf{else}\ \mathbf{return}\ \mathit{false}\ \mathbf{fi} \\
\mathit{LL}(X) & \equiv & \mathit{valid}_X[p] := \mathit{true};\ \mathbf{return}\ X \\
\mathit{SC}(X, v) & \equiv & \mathbf{if}\ \mathit{valid}_X[p]\ \mathbf{then} \\
 & & \quad X := v; \\
 & & \quad \mathbf{for}\ i := 0\ \mathbf{to}\ N - 1\ \mathbf{do}\ \mathit{valid}_X[i] := \mathit{false}\ \mathbf{od}; \\
 & & \quad \mathbf{return}\ \mathit{true} \\
 & & \mathbf{else} \\
 & & \quad \mathbf{return}\ \mathit{false} \\
 & & \mathbf{fi} \\
\mathit{VL}(X) & \equiv & \mathbf{return}\ \mathit{valid}_X[p]
\end{array}
$$

Figure 3.1: Equivalent atomic code fragments for common instructions used. Fragments for LL, VL, and SC 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 previously executed a LL instruction.

stress that these code fragments are definitions, and should not be interpreted as implementations of the given operations. The test-and-set instruction atomically sets a bit to *true* and returns *true* if it changes the bit from *false* to *true*, and *false* otherwise. The fetch-and-add instruction atomically adds a value to a variable and returns the previous value of that variable. The CAS operation takes three parameters — a variable, an old value, and a new value. If the variable equals the old value, then CAS writes the new value to the variable and returns *true*. Otherwise, it returns *false* and does not modify the variable. The LL operation returns the value of the variable on which it is invoked. A *successful* SC operation on variable X writes a new value to X and returns *true*. An *unsuccessful* SC operation on X returns *false* and does not modify X. A SC operation by process p on variable X is successful if no process has performed a successful SC on X since p's most recent LL on X, and is unsuccessful otherwise. A validate (VL) instruction is sometimes associated with the

LL/SC pair. A VL instruction by process p on variable X returns *true* if no process has performed a successful SC on X since p's most recent LL on X, and returns *false* otherwise.

Shared memory is implemented differently in various multiprocessors. For example, some multiprocessors arrange all processors and memory on a single shared bus, while others provide multiple processor-to-memory paths by using several buses or a crossbar. However, for the most part, these implementation details do not concern us. (An exception arises in Chapter 6, where, for performance reasons, we present different algorithms for machines with cache-coherence mechanisms and for distributed shared memory machines on which different parts of shared memory can be accessed at different speeds.) However, we do assume that the shared memory provides sequential consistency [63], which, roughly speaking, ensures that all memory operations appear to occur in the same total order to all processors.

We model computations on shared memory multiprocessors using states and histories. A *state* is a mapping that assigns a value to all shared memory locations and to all private process variables (including the program counter for each process). A *history* is a totally-ordered sequence of states, with each pair of consecutive states separated by a process *step* that causes the computation to go from the first state to the second. Thus, in the history $s_0 \xrightarrow{p_1} s_1 \xrightarrow{p_2} s_2 \xrightarrow{p_3} \ldots$, the initial state is s_0; the execution of process p_1's first statement causes the computation to go from state s_0 to state s_1; the execution of process p_2's first statement causes the computation to go from state s_1 to state s_2, and so on.

When considering a collection of processes that share an object, we model the processes as infinite loops that repeatedly execute a *remainder section* (unrelated computation) and invoke operations on the object. This model is in keeping with the programming paradigm described in Chapter 1. The *level of contention* is the number of processes that are concurrently outside their remainder sections (that is, the number of processes that concurrently access the shared object). A shared object implementation is *scalable under contention* if its performance does not degrade unacceptably as the level of contention increases. The notions of an implementation being scalable with increasing object size or scalable with an increasing number of processes or processors are defined similarly.

In order to model asynchronous processes with arbitrary delays (and even failures), we assume the existence of a scheduler that repeatedly selects and executes a step of any process that is ready to run. It is important to note that this is not a real scheduler: it is an abstraction that we use to model execution interleavings. (However, this abstraction is sufficiently general to capture the behavior of real schedulers.) We prove that our algorithms are correct for any history generated by this scheduler. Some of our correctness properties refer to faulty and nonfaulty processes. A process p is *faulty* in a history h if, after some point, h contains no steps of process p; p is *nonfaulty* otherwise. (We sometimes consider a process to be faulty only if it stops taking steps when executing within a certain section of its code. In this case, we say a process p is faulty in a history h if and only if there is some point in h at which p is executing within a certain section of its code, and after which no step by process p occurs.)

Below we describe the proof techniques used throughout the dissertation. First, however, a brief discussion of our programming notation, which should be largely self-explanatory, is in order. As an example, consider Figure 3.2. (This algorithm is explained

```
shared variable X : array[0..k − 1, 0..k − 1] of {⊥} ∪ {0..N − 1};
                Y : array[0..k − 1, 0..k − 1, 0..k] of {⊥} ∪ {0..N − 1}
initially (∀r, c, n : 0 ≤ r < k − 1 ∧ 0 ≤ c < k − 1 ∧ r + c < k − 1 ∧ 0 ≤ n ≤ k − r − c :: Y[r, c, n] = ⊥)

private variable name: 0..k(k + 1)/2 − 1;  moved: boolean;  i, j: 0..k − 1;  h: 0..k
initially i = 0 ∧ j = 0 ∧ h = 0

    while true do
0:    Remainder Section;
      i, j, moved := 0, 0, true;
      while i + j < k − 1 ∧ moved do
1:      X[i, j], h, moved := p, 0, false;
        while h < k − i − j ∧ ¬moved do
2:        if Y[i, j, h] ≠ ⊥ then
            j, h, moved := j + 1, 0, true
          else h := h + 1
          fi
        od;
        h := 0;
        while h ≤ k − i − j ∧ ¬moved do
3:        if X[i, j] = p then
4:          Y[i, j, h], h := p, h + 1
          else
            while h > 0 do
5:            h := h − 1;
              if Y[i, j, h] = p then
6:              Y[i, j, h] := ⊥
              fi
            od;
7:          moved, i := true, i + 1
          fi
        od
      od;
8:    name := ik − i(i − 1)/2 + j;
9:    Working;
      if ¬moved then
        while h > 0 do
10:       h := h − 1;
          if Y[i, j, h] = p then
11:         Y[i, j, h] := ⊥
          fi
        od
      fi
    od
```

Figure 3.2: An example of our programming notation.

in Chapter 7 and proved correct in Appendix C; it is used here only for explanation.) In this and other programs (unless stated otherwise), the code starting at one label and continuing until another label is encountered is assumed to be atomic. In figures in which no labels are given, each line of code is assumed to be atomic. To simplify our proofs, we sometimes label somewhat lengthy code fragments. For example, the execution of the code starting at label 2 in Figure 7.6 is assumed to read $Y[i, j, h]$, modify j, h, and *moved* accordingly, check the loop condition before label 2, and if that fails, assign zero to h, check the loop condition at label 3, and if that fails, check the loop condition at label 1, and set the program counter to 2, 3, 1, or 8, accordingly. Nonetheless, this and other atomic program fragments each access at most one shared variable; it is therefore easy to implement them atomically using the assumed memory operations in each case. Finally, private variables in all figures are assumed to retain their values between procedure calls.

As discussed in Section 2.2.1, the correctness condition we use for our lock-free and wait-free implementations is linearizability. An implementation of an object is *linearizable* if, in every history h, the partial order over the operation invocations in h can be extended to a total order such that the sequence of operations in the total order is consistent with the sequential semantics of the implemented operations. In our linearizability proofs, we show that this total order exists by defining, for each operation invocation, a unique point in time, called the *linearization point* of that invocation. (This point may occur at different points in different invocations of the same operation.) We say that an invocation is *linearized to* its linearization point. We also define a "current" value of the implemented variable. We then show that, at the linearization point of each invocation, the value of the implemented variable before and after that point is consistent with the semantics of the implemented operation, and that the invocation returns the same value as the sequential operation would if executed atomically at that linearization point.

Assertional proof techniques [27] are used to prove correct the more difficult algorithms in this dissertation. We prove that an assertion I is an invariant by showing that it holds inductively or that it follows from established invariants. For an inductive proof, we show that I holds initially and that I is not falsified[1] by any statement execution, i.e., if I (and perhaps other established invariants) holds before a given statement is executed then I holds afterwards. When an invariant is in the form of an implication, we often achieve this by showing that the consequent holds after the execution of any statement that establishes the antecedent, and that no statement falsifies the consequent while the antecedent holds. We prove assertions of the form P *unless* Q by proving for every state s of every history h, if $P \wedge \neg Q$ holds in state s, then $P \vee Q$ holds in the subsequent state in h. Thus, P *unless* Q implies that, in every history, if P holds in a state of that history, then P is stable in that history until Q holds. (If Q does not hold at or after that state, then P holds forever.) Finally, P *leads-to* Q holds if, for every history h, and for every $i \geq 0$, if P holds in state s_i of h, then there exists a $j \geq i$, such that Q holds in state s_j of h.

Most of the progress properties in this dissertation are quite straightforward. We model fairness in terms of the faulty and nonfaulty processes discussed above. The following notational conventions and proof methods are used throughout this dissertation to state properties about our algorithms, and to prove them correct.

[1]A statement execution *falsifies* an expression if and only if that expression holds before the statement execution and does not hold afterwards. Similarly, a statement execution *establishes* an expression if and only if that expression does not hold before the statement execution, but holds after.

Notational Conventions: In all of our algorithms and proofs, the number of processes is denoted by N, and process labels p, q, r, and s range over $0, ..., N-1$. In each algorithm presented, the code is given for process p. Other free variables are assumed to be universally quantified. We use $P^{x_1, x_2, \ldots, x_n}_{y_1, y_2, \ldots, y_n}$ to denote the expression P with each occurrence of x_i replaced by y_i. The predicate $p@s$ holds if and only if statement s is the next statement to be executed by process p. We use $p@S$ as shorthand for $(\exists s : s \in S :: p@s)$, $p.s$ to denote statement s of process p, and $p.var$ to denote p's local variable var. The following is a list of symbols we use in our proofs, in increasing order of binding power: $\equiv$, $\Rightarrow$, $\vee$, $\wedge$, $(=, \neq, <, >, \leq, \geq)$, $(+, -)$, (multiplication,/), $\neg$, $(., @)$, $(\{, \})$. Symbols in parentheses have the same binding power. We occasionally use parentheses to override these binding rules. We sometimes use Hoare triples [49] to denote the effects of a statement execution. $\Box$

Chapter 4

Support for Algorithms and Related Results

In this chapter, we present several implementations involving universal primitives. These implementations allow the LL/SC-based constructions presented in the following chapters to be applied with greater flexibility. In particular, we present time-optimal, wait-free implementations of LL and SC from Read and CAS, and vice versa, and implementations that eliminate the need to deal with spurious SC failures. We also present an implementation of LL and SC operations that access variables that are larger than a single machine word. As explained below, these new constructions simplify the design of algorithms in later chapters, as well as future algorithms, by allowing us to ignore certain limitations of the available universal primitives.

The algorithms presented in Chapter 5 depend on the availability of LL, VL, and SC operations that access variables that are larger than the word size on most multiprocessors. This presents several potential difficulties for using these algorithms in practice. First, some machines do not provide LL, VL, and SC operations, but do provide other universal primitives, such as CAS. Second, the LL and SC instructions are frequently implemented on top of a cache coherence mechanism. In this approach, a LL operation by process p loads the variable accessed into p's local cache, and a SC operation modifies that variable only if it still resides in the cache. If another process performs a successful SC operation between p's LL and SC, then the coherent cache mechanism invalidates p's cache copy of the variable. Thus, p's SC subsequently fails, in keeping with the semantics of LL and SC. However, this approach introduces the possibility of a SC operation failing as a result of an unrelated cache invalidation, even though the semantics of LL and SC do not dictate that it should fail. This "spurious" behavior complicates algorithm design, and can lead to incorrect behavior if not addressed carefully. We call a SC operation that is subject to spurious failures FSC. Finally, LL and SC usually access only one machine word, which precludes their use in some algorithms.

We present several key results in this chapter that overcome these difficulties, making our algorithms and others easier to apply in practice. First, we present constant-time implementations of LL, VL, and SC operations using Read, Write, and CAS operations. This allows algorithms (including ours) that are based on LL, VL, and SC operations to be implemented on machines that provide the CAS operation. For completeness, we also present a constant-time implementation of Read, Write, and CAS using LL and SC operations. These results show that CAS is equivalent to LL and SC from a performance standpoint — this stands in contrast to the commonly-held belief that LL and SC necessarily result in more efficient object implementations. Next, we present an implementation of LL, VL, and SC from LL and FSC. This result allows algorithm designers to ignore the issue of spurious failures.

Finally, we show how to efficiently implement LL, VL, and SC for multi-word variables using the usual single-word LL, VL, and SC primitives.[1] In this implementation, LL may return a special "failure" value that indicates that a subsequent SC will fail — we call this a *weak*-LL. Elsewhere [10], we present a similar construction, in which the LL operation has the "normal" semantics: it always returns the current value of the implemented variable, even if the subsequent SC operation is sure to fail. The ability to return a special failure value results in a simpler construction with lower space requirements 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 performing private updates when a subsequent SC is certain to fail). For these reasons, we use weak-LL in the universal constructions presented in the next chapter. Time complexity bounds for the implementations presented in this chapter are summarized in Table 4.1.

Although existing universal constructions can be used to convert between CAS and LL, VL, and SC, and to implement multi-word LL and SC, such constructions entail high overhead because they treat these operations as general shared objects, and are therefore

[1]The multi-word operations considered here access a *single* variable that spans multiple contiguous words. Thus, they are not the same as the multi-word operations considered in [9, 18, 54, 84], which access *multiple* variables that are stored in separate, and not necessarily contiguous, memory words. The multi-word operations we consider admit simpler and more efficient implementations than those considered in [9, 18, 54, 84].

[1]The SC operation terminates in $\Theta(1)$ time after the most recent spurious FSC. See Section 4.1.3 for details.

Primitives Used	Primitives Implemented	Time Complexity
Read, CAS	LL, VL, SC	$\Theta(1)$, $\Theta(1)$, $\Theta(1)$
LL, SC	Read, CAS	$\Theta(1)$, $\Theta(1)$
LL, FSC	LL, VL, SC	$\Theta(1)$, $\Theta(1)$, $\Theta(1)^2$
LL, VL, SC	W-word weak-LL, VL, and SC	$\Theta(W)$, $\Theta(1)$, $\Theta(W)$

Table 4.1: Summary of results concerning synchronization primitives. The first three lines give the time complexity of implementations of one-word variables. The last line gives the time complexity of an implementation for W-word variables.

unable to improve efficiency by taking advantage of the semantics of the implemented operations. In contrast, our implementations of these primitives are time-optimal. The best previous wait-free implementation of LL, VL, and SC using Read and CAS, recently presented by Israeli and Rappoport in [54], requires $\Theta(N)$ time per operation. It also requires N-bit shared variables, which severely limits its usefulness in practice. (Israeli and Rappoport did not present similar constructions for CAS, and, to our knowledge, the other constructions in Table 4.1 have not previously been considered.)

4.1 One-Word Primitives

In this section, we present efficient implementations of one-word synchronization primitives. We begin with a constant-time implementation of LL, VL, and SC using Read and CAS.[3] We then present a simple, constant-time implementation of Read and CAS from LL and SC. The latter construction assumes that SC does not fail spuriously. We conclude this section by using LL and FSC to implement LL and SC. This result allows us to use our constructions in systems where SC might fail spuriously. To avoid confusion, we refer

[3]More accurately, we use shared registers that support atomic Read and Write operations, as well as shared registers that support Read and CAS operations. We similarly assume the availability of read/write registers in subsequent constructions.

type	$llsctype =$ **record** $value$: $valtype$; tag: $0..2N+1$; pid: $0..N-1$ **end**
shared variable	X: $llsctype$; A: **array**$[0..N-1]$ **of** $llsctype$
initially	$X =$ (initial value of implemented variable, $0, 0$)
private variable	old, chk: $llsctype$; j: $0..N-1$; $newtag$: $0..2N+1$
initially	$j = 0$

procedure $LL()$
1: $old := X$;
2: $A[p] := old$;
3: $chk := X$;
4: **return** $old.value$

procedure $VL()$
5: **return** $chk = old \wedge X = old$

procedure $SC(val\text{: } valtype)$
6: **if** $chk \neq old$ **then return** *false* **fi**;
7: **read** $A[j].tag$;
8: $j := (j+1)$ **mod** N;
9: **select** $newtag : newtag \notin$ {last N tags read by p} $\cup$
{last N tags selected by p} $\cup$
{last tag successfully CAS'd by p};
10: **return** $CAS(X, old, (val, newtag, p))$

Figure 4.1: Constant-time LL, SC, and VL using Read and CAS. Private variables are static between invocations.

to the implemented LL in the latter construction as safe-LL. We also refer to the variable that supports the implemented operations as the *implemented variable* to avoid confusion with the variables used in the implementation.

4.1.1 Implementation of LL/SC using CAS

Figure 4.1 depicts an N-process implementation of LL, VL, and SC that is based on Read and CAS.[4] This algorithm uses a nondeterministic choice to select a new value — subject to the constraints indicated — for *newtag* at line 9. The algorithm is correct for any implementation of this tag selection mechanism that satisfies these constraints. After describing the algorithm and proving it correct, we describe a simple, constant-time implementation of this tag selection mechanism.

[4]Here, as well as in other implementations presented in this chapter, we present the algorithms and associated variables for implementing the stated instructions on *one* variable. For this reason, the name of the implemented variable is omitted from the parameters of the procedure calls (for example, the SC procedure given in Figure 4.1 takes only one parameter — the value to be written). It would be straightforward to use Read and CAS to implement multiple variables, each supporting LL, VL, and SC operations, by duplicating the variables in Figure 4.1, and by establishing an appropriate naming scheme to distinguish between the implementations.

Variable X contains the value of the implemented variable, along with a tag and process identifier (the purpose of these fields is explained below). The basic structure of this algorithm is that a LL operation by process q records the value of X in $q.old$ and returns that value (lines 1 and 4), and SC uses CAS to attempt to change X from the value stored in $q.old$ to a new value val (line 10). Process q's SC is successful if and only if the CAS is successful. If the value of X changes between q's executions of lines 1 and 10, (i.e., some other process performs a successful SC in this interval), then q's CAS (and therefore the implemented SC) should fail, in keeping with the semantics of LL and SC. Similarly, if X does not change in this interval (i.e., no successful SC is performed in this interval), then q's CAS (and therefore the implemented SC) succeeds. Again, this is consistent with the semantics of LL and SC. However, if the value of X changes more than once between q executions of lines 1 and 10, or if some process performs a successful SC that does not modify X (i.e., it writes the same value that is already in X), then there is a risk that q's CAS succeeds, because X contains the same value that p's LL operation read. However, the semantics of LL and SC dictate that q's SC should *fail* in this case.

The *pid* and *tag* fields that are stored with X are used to prevent the error described above from arising. This is achieved by having processes store a tag with each value written to X, and by introducing a feedback mechanism between processes that ensures that the value of X (including the *pid* and *tag* fields) read by the LL procedure of a process q is not written to X again before q's CAS. This feedback mechanism consists of the value written by q to $A[q]$ at line 2 and the tag values read by other processes at line 7.

The key property in proving this implementation correct, which is formalized by the following claim, is that a process p does not prematurely reuse a tag, thereby causing a CAS by some process q to succeed when it should fail.

Claim 1: If process q reads (x, v, p) from X at both line 1 and line 3 in its LL procedure, then process p does not select tag v after q's second read of X and before q's CAS.

Proof: Observe that $X = (x, v, p)$ holds at q's second read of X and that $A[q] = (x, v, p)$ holds between q's second read of X and q's CAS. Suppose p selects tag v in this interval. Because p does not use any of the N most recently selected tags, it follows that p performs at least N SC operations before selecting tag v again. Thus, because there are N processes, and because p reads $A[r]$ for a different r in each SC operation (see lines 7 and 8), p must have read $A[q] = (x, v, p)$ in the last N operations, and therefore does not select tag v. $\square$

Linearizability: For the algorithm in Figure 4.1, we define the current value of the implemented variable to be $X.value$. We further define the linearization point of a LL operation to occur at the operation's first read of X (line 1) if the values read from X at lines 1 and 3 differ, and at its second read of X (line 3) otherwise. A SC is linearized to occur at line 6 if it returns from line 6, and at its CAS otherwise. A VL is linearized to occur when it reads X. For the purposes of the linearizability proof, we define a VL or SC operation to be *failed* if it returns *false* and *successful* otherwise.

First, note that $X.value$ changes only as a result of a successful CAS, and that, by definition, each successful CAS corresponds to the linearization point of a successful SC operation. Thus, $X.value$ always contains the correct value of the implemented variable (i.e., the value written by the most recent successful SC operation). It is easy to see that the value returned by a LL operation is the current value of the implemented variable at that operation's linearization point. Also, by definition, failed VL and SC operations return *false*, and successful VL and SC operations return *true*. It remains to show that VL and SC operations correctly fail or succeed at their linearization points according to the sequential semantics of these operations.

If a VL or SC operation by process q returns *false* because $q.chk \neq q.old$, then a successful CAS is performed (and therefore a successful SC is linearized) after q's first read of X (which is also the linearization point of q's LL in this case). Thus, q's operation is correctly linearized in this case. Similarly, if a SC or VL operation by process q returns *false* because $X \neq q.old$, then a successful SC operation is linearized after q's second read of X (which is also the linearization point of q's LL in this case). Again, q's operation is correctly linearized. It remains to consider the case in which a VL or SC operation by process q returns *true*. In this case, Claim 1 implies that no successful SC operation is linearized after the second read of X in q's previous LL, which is the linearization point of q's LL (note that both VL and SC return *false* if $q.chk \neq q.old$). Thus, q's VL or SC operation is correctly linearized. $\square$

Below we describe a mechanism that allows each new tag to be selected in constant time using $\Theta(N)$ space per process. Given this mechanism, the proof above, and a straightforward time and space complexity analysis, we have the following result.

Theorem 1: LL, VL, and SC can be implemented with constant time complexity and $\Theta(N^2)$ space complexity using Read and CAS. $\square$

The algorithm in Figure 4.1 can be implemented with a variety of tag selection mechanisms. For example, if each process chooses a previously-unused tag in every execution of SC, then the tag selection criteria in line 9 are met. This would result in the tags growing without bound. However, in applications where SC would not be called sufficiently many times to cause an overflow, this does not pose a practical problem. For completeness, we now describe a tag selection mechanism that does not require unbounded tags.

In our bounded tag selection mechanism, each process maintains three local queues — *Read*, *Last*, and *Select*. (Note that these are *not* concurrent queue implementations, and are therefore easily implemented using standard data structures.) The *Read* queue contains the last N tags read. The *Last* queue contains a single tag, which is the last one successfully written (using CAS) to X. The *Select* queue, from which new tags are selected, contains all tags that are not in the *Read* queue or the *Last* queue.

The tag queues are maintained by means of the *Read_Tag*, *Store_Tag*, and *Select_Tag* procedures shown in Figure 4.2. In these procedures, *enqueue* and *dequeue* denote the normal queue operations, $delete(Q, v)$ removes tag v from Q (and does not modify Q if v is not in Q), and $x \in Q$ holds if and only if tag x is in queue Q.

To allow a correct tag to be selected at line 9, the *Read_Tag*, *Store_Tag*, and *Select_Tag* procedures are incorporated into the algorithm in Figure 4.1. Specifically, when process p reads a tag from $A[j]$ at line 7, p calls *Read_Tag* to record the tag it reads; process p calls *Select_Tag* at line 9 to select a new tag; and process p calls *Store_Tag* at line 10 after a successful CAS to record the tag successfully stored in X. We now describe these procedures, and explain how they guarantee that the tag selected at line 9 satisfies the

```
private variable Read_Q, Select_Q, Last_Q: queue of 0..4N + 1;   y: 0..4N + 1
initially        Last_Q={0};  Read_Q={1, ..., N};  Select_Q={N + 1, ..., 4N + 1}

procedure Read_Tag(v)
  if v ∈ Read_Q then
    delete(Read_Q, v);
    enqueue(Read_Q, v)
  else
    enqueue(Read_Q, v);
    delete(Select_Q, v);
    y := dequeue(Read_Q);
    if y ∉ Last_Q then
      enqueue(Select_Q, y)
  fi fi

procedure Store_Tag(v)
  delete(Select_Q, v);
  enqueue(Last_Q, v);
  y := dequeue(Last_Q);
  if y ∉ Read_Q then
    enqueue(Select_Q, y)
  fi

procedure Select_Tag()
      returns 0..4N + 1
  y := dequeue(Select_Q);
  enqueue(Select_Q, y);
  return y
```

Figure 4.2: Pseudo-code implementations of operations on tag queues.

required constraints.

Select_Tag moves the front tag in p's *Select* queue to the back, and returns that tag. *Store_Tag* moves the tag from the *Select* queue to the *Last* queue, removes the tag that was previously in the *Last* queue, and, if that tag is not in the *Read* queue, returns it to the *Select* queue.

Read_Tag records that a tag v was read as follows. If v is already in p's *Read* queue, then *Read_Tag* simply moves v to the end of the p's *Read* queue. If v is not already in p's *Read* queue, then *Read_Tag* enqueue v into p's *Read* queue and removes it from p's *Select* queue (assuming v is in p's *Select* queue). Finally, *Read_Tag* removes the tag at the front of p's *Read* queue because it is no longer one of the last N tags read by p. If that tag is also not the last tag written to X by p, then *Read_Tag* returns it to p's *Select* queue.

Process p's *Read* queue always contains the last N tags read by process p, and the *Last* queue always contains the last tag successfully written to X by p. Thus, the tag selected by *Select_Tag* is not the last tag successfully written to X by p, nor is it among the last N tags read by p. In fact, maintaining a total of $4N + 2$ tags ensures that the tag selected is also not one of the last N tags selected. (In principle, only $2N+1$ tags are required, and this value is used in Figure 4.1 and other figures that use this mechanism. Our queue-based implementation of the tag selection mechanism provides constant-time operations, but requires $4N+2$ tags.)

To see why $4N+2$ tags suffice in Figure 4.2, first note that p's *Read* queue always contains N tags, and p's *Last* queue always contains one tag (except temporarily during the execution of *Read_Tag* or *Store_Tag*). Thus, using $4N+2$ tags ensures that there are always at least $3N+1$ tags in p's *Select* queue. (It is possible that the tag in p's *Last* queue is also in p's *Read* queue, which results in there being $3N+2$ tags in p's *Select* queue, rather than $3N+1$.) Thus, when a tag v is selected by process p, and *Select_Tag* moves v to the back of p's *Select* queue, there are at least $3N$ tags ahead of v in p's *Select* queue. This implies that v is not selected again by process p until $3N$ tags are removed from p's *Select* queue. At most three tags are removed from p's *Select* queue per SC operation by process p (one by *Read_Tag*, one by *Select_Tag*, and one by *Store_Tag*). Thus, tag v is not selected again by process p until at least N operations that remove tags from p's *Select* queue have been executed by process p. Also, it is easy to see that, if an operation by process p removes any tags from p's *Select* queue, then that operation selects a tag. Thus, the tag selected by process p is never one of the last N tags selected by process p, as required.

All of the queue operations described above can easily be implemented in constant time. This is achieved by using a static array that is indexed by tag and contains, for each tag, a pointer to the queue node containing that tag. This allows the queue node for a given tag to be located in constant time. Also, representing the queues as doubly-linked

shared variable X: *valtype*

procedure CAS(*old*, *new*: *valtype*)
1: **if** $LL(X) \neq old$ **then return** *false* **fi**;
2: **if** $old = new$ **then return** *true* **fi**;
3: **return** $SC(X, new)$

Figure 4.3: Constant-time implementation of CAS using LL and SC. LL trivially implements Read.

lists allows that node to be removed from a queue or added to a queue in constant time.

4.1.2 Implementation of CAS using LL/SC

We now turn our attention to the implementation of Read and CAS using LL and SC shown in Figure 4.3. The current value of the implemented variable is defined to be the current value of X. Read is trivially implemented by performing a LL on X and returning the value returned by the LL. Process p performs a CAS by reading variable X using LL, and possibly performing a subsequent SC of X. We define the linearization point of a Read operation to be the point at which that operation executes the LL on X. (Clearly the value returned by the Read operation is the current value of the implemented variable when the LL is executed; the Read operation is therefore correctly linearized at that point.) We now show that every CAS operation can be correctly linearized.

If a CAS operation by process p returns from line 1 or from line 2, then p's CAS can be linearized to the point at which the LL occurs. To see why, observe that, in the first case, p's operation returns *false*, and the sequential semantics of CAS dictates that the operation should fail because the implemented variable differs from p's *old* value. In the second case, the implemented variable equals p's *old* value immediately before the LL, and equals p's *new* value immediately after the LL. Thus, linearizing p's CAS to the point at which the LL is executed is consistent with the sequential semantics of a successful CAS, and p's operation returns *true* in this case. It remains to consider CAS operations that return from line 3.

If the SC at line 3 is successful, then, because the LL at line 1 reads $X = p.old$ (otherwise the CAS returns from line 1), the value of the implemented variable equals p's *old* value immediately before the SC operation is executed (if X changes between the LL and the SC, the SC fails). Also, the SC writes p's *new* value to X. Thus, because the CAS operation returns *true* in this case, linearizing the CAS to occur at the execution of the SC is consistent with the sequential semantics of CAS.

If the SC at line 3 fails, then a successful SC by another process has occurred since p's previous LL. Because each successful SC changes the value of X (note that, if $q.old = q.new$, then CAS returns from line 1 or line 2), there is a point during p's CAS at which X differs from *old* (either before or after the SC that changes the value of X); p's (failed) CAS is correctly linearized at that point because it returns *false* in this case. The time and space complexity analysis for this algorithm is straightforward, so we have the following theorem.

Theorem 2: Read and CAS can be implemented with constant time and space complexity using LL and SC. □

We should point out that ordinary Read and Write operations are straightforward to incorporate into the constructions of Figures 4.1 and 4.3. In particular, because the current value of the implemented variable is stored in X in each case, Read can be implemented in the construction of Figure 4.1 by reading X, and in the construction of Figure 4.3 by performing a LL on X. Also, $Write(new)$ can be implemented in the construction of

```
type              tagtype: record value: valtype; tag: 0..2N + 1; pid: 0..N − 1 end
shared variable   X: tagtype; A: array[0..N − 1] of tagtype
private variable  old, chk: tagtype; j: 0..N − 1; newtag: 0..2N + 1
initially         j = 0

procedure Safe_LL()                  procedure SC(new: valtype)
    old := X;                            if chk ≠ old then return false fi;
    A[p] := old;                         read A[j].tag;
    chk := LL(X);                        j := (j + 1) mod N;
    return old.value                     select newtag : newtag ∉ {last N tags read}∪
                                                                  {last N tags selected}∪
                                                                  {last tag successfully FSC'd};
procedure VL()                           while true do
    chk := LL(X);                            if FSC(X, (new, newtag)) then return true
    return old = chk                         elseif LL(X) ≠ old then return false
                                             fi
                                         od
```

Figure 4.4: Implementation of LL, SC, and VL using LL and FSC.

Figure 4.1 like SC; the main difference is that X is updated by a Write rather than a CAS. *Write*(new) can be implemented in the construction of Figure 4.3 by the following code, which is similar to that given for CAS.

$$\textbf{if } LL(X) = new \textbf{ then return fi};$$

$$SC(X, new)$$

4.1.3 Implementation of LL/SC using LL/FSC

In Figure 4.4, we present an implementation of safe-LL,[5] VL, and SC from LL and FSC. The basic structure of this implementation is that LL implements safe-LL and FSC implements SC. The only complication is that FSC might fail spuriously, and we use a tag-based feedback mechanism similar to the one used in Figure 4.1 to allow a process to identify a spurious FSC failure and to retry the FSC. This is achieved by writing tags with each new value stored to X. As in the algorithm in Figure 4.1, we avoid reusing a tag

[5]We call the implemented operation safe-LL in Figure 4.4 to distinguish it from the LL instruction used in the implementation.

that has been read by a safe-LL of some process that has not yet performed a subsequent SC. Thus, when a SC by process p fails, p can reread X to see if it has changed. If X has changed, then another process has performed a successful SC, so p can correctly fail and return *false*. If the X value read is equal to the value read previously, then the feedback mechanism guarantees that X has not been modified since p's LL, which implies the SC failed spuriously. In this case, the FSC can be retried.

Observe that, when an FSC is retried in this construction, the FSC instruction is executed immediately after the LL instruction. This is important for two reasons. First, it implies that the "window of vulnerability", in which cache invalidations (and potentially subsequent spurious failures) can occur, is quite short. Secondly, there are no shared variable references in this interval. This further decreases the likelihood of subsequent FSC executions failing spuriously. Furthermore, some hardware implementations of the LL and SC instructions forbid processes to access shared memory between the execution of LL and the subsequent execution of SC. The construction in Figure 4.4 can easily be modified so that this rule is not violated. This is achieved by using Read instead of LL in the implementations of safe-LL and VL, and by changing the implementation of SC so that the LL check is always performed before the FSC, rather than being performed only in the event of a FSC failure. (It would also be necessary to ensure that the *old* variable is stored in a register, so that the check between the LL and FSC does not access memory.) This allows algorithms that do access shared memory between the LL and SC operations to be implemented on architectures that forbid it.

It is easy to see that, if FSC does not fail infinitely often during one invocation of the implemented SC, then the implemented SC eventually terminates. The linearizability proof for this construction is almost identical to that of the construction given in Figure 4.1, and is therefore omitted. Given the tag selection mechanism presented earlier, the time and space complexity analysis for this algorithm is straightforward, giving the following theorem.

Theorem 3: With space complexity $\Theta(N^2)$, LL and FSC can be used to implement constant-time safe-LL and VL operations, and a SC operation that, provided only finitely many spurious FSC failures occur per SC invocation, terminates in constant time after the last spurious failure. □

4.2 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.[6] 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 Chapter 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 ID of some process that performed a successful SC

[6]We assume that the SC operation does not fail spuriously. As shown in Section 4.1.3, a SC operation that does not fail spuriously can be efficiently implemented using LL and a SC operation that might fail spuriously.

```
shared var 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 var curr: record pid: 0..N − 1; tag: 0..1 end; i: 0..W − 1; side: 0..1
initially side = 0

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

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

Figure 4.5: W-word weak-LL and SC using 1-word LL, VL, and SC. W-word VL is implemented by validating X.

during the execution of the weak-LL operation. We call the process whose ID is returned a *witness* of the failed weak-LL. As we will see in Section 5.2, the witness of a failed weak-LL can provide useful state information that held during the execution of that weak-LL.

We now describe our implementation of weak-LL, VL, and SC, which is shown in Figure 4.5.[7] 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 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. 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 by 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

[7]Recall that private variables in all figures are assumed to retain their values between procedure calls.

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.

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.) If the VL fails, then the weak-LL re-reads X at line 4 in order to determine the ID of the last process to perform a successful SC; this process ID is then returned. Note that if the VL of line 3 fails, then a subsequent SC by p will fail. In Appendix A, we prove this construction correct in the context of an algorithm from Chapter 5 that uses it. The algorithm in Figures 4.5 yields the following result.

Theorem 4: Weak-LL, VL, and SC operations for a W-word variable can be implemented using LL, VL, and SC operations for a one-word variable with time complexity $\Theta(W)$, $\Theta(1)$, and $\Theta(W)$, respectively, and space complexity $\Theta(NW)$. □

Chapter 5

Large Objects

In this chapter, 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. The constructions in this chapter are based on LL, VL, and SC operations for a multi-word shared variable. As shown in the previous chapter, these operations can be efficiently implemented from similar one-word primitives.

This chapter extends recent research on universal lock-free and wait-free constructions of shared objects [43, 44]. Such 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, this generality often comes at a price, specifically space and time overhead that is excessive for many objects. In this chapter, we address these shortcoming by presenting more efficient universal constructions that can usually be used to implement large objects with low space overhead.

We take as our starting point the lock-free and wait-free universal constructions for small objects presented by Herlihy in [44]. 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 LL operation, and a private copy of the object is made; then, the desired operation is performed on the private copy; finally, a SC operation is executed to attempt to modify the shared object pointer to point to the private copy. The SC operation may fail, in which case these steps are repeated. This algorithm is lock-free because a SC operation by process p fails only if another SC has succeeded since p's most recent LL. This implies that, if p does not complete its operation, then some other process does. However, the algorithm is not wait-free because the SC of each loop iteration of a particular process 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, Herlihy shows that it is helped after at most two loop iterations [44]).

As Herlihy points out, these constructions perform poorly if used to implement large objects. To overcome this problem, he presents a lock-free construction in which a large object is fragmented into blocks linked by pointers. In this construction, operations are implemented so that only those blocks that must be accessed or modified are copied.

Herlihy's lock-free approach for implementing large objects suffers from three shortcomings. First, the fragmentation technique used often requires a significant amount of creative work on the part of the sequential object designer before the advantages of Herlihy's large-object construction can be realized. In particular, the programmer must determine how the object should be fragmented based on its semantics, and must also include code in each operation that explicitly copies parts of the object in order to avoid interference by a concurrent operation of another process. Second, Herlihy's approach is difficult to apply in wait-free implementations. In particular, directly combining it with the helping mechanism of his wait-free construction for small objects results in excessive space overhead. Third, Herlihy's large-object techniques reduce copying overhead only if long "chains" of linked blocks are avoided. Consider, for example, the implementation of a large shared queue. Given the blocks-and-pointers structure dictated by Herlihy's large-object construction, it is natural to implement the queue as a linear sequence of blocks (i.e., in a linked list). An unfortunate consequence of this approach is that adding a new block to the list (for an *enqueue* operation) actually requires the replacement of every block in the list. In particular, linking in a new last block requires that the pointer in the previous block be changed. Because that block is modified, it must be copied to a new location to avoid interference from concurrent operations. Thus, the pointer in the next-to-last block must be modified, so the next-to-last block must also be replaced. Repeating this argument, it follows that every block in the list must be replaced.

Our approach for implementing large objects is also based upon the idea of fragmenting an object into blocks. However, our large-object constructions provide the object programmer with a more natural programming paradigm than Herlihy's does. Specifically, our constructions allow the programmer to treat the object as if it were stored in contiguous locations in shared memory, while Herlihy's construction dictates a blocks-and-pointers approach. Thus, we view a large object as a long array that is fragmented into blocks. However, unlike Herlihy's approach, the fragmentation in our approach is not visible to the object programmer. Also, copying overhead in our approach is often much lower than in Herlihy's. For example, we can implement shared queues with constant copying overhead.

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, VL, and SC operations that access a "large" shared variable that contains all block pointers. This large variable is stored across several memory words, and is accessed using the LL, VL, and SC operations for large variables presented in the previous chapter.

Our wait-free universal construction is the first such construction to incorporate techniques for implementing large objects. In this construction, we impose an upper bound on the number of blocks each process has for copying. This bound is assumed to be large enough to accommodate any single operation. 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 operation help as many processes as the space constraints allow, 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 present our lock-free universal construction in Section 5.1, our wait-free universal construction in Section 5.2, and the results of performance experiments comparing our constructions to Herlihy's in Section 5.4. Finally, a full assertional correctness proof for the wait-free construction is presented in Appendix A.1.

5.1 Lock-Free Universal Construction for Large Objects

Our lock-free construction is shown in Figure 5.1. This construction provides the object programmer with a general framework for the implementation of shared objects by enabling her to treat the object as if it were stored in a contiguous array. Unlike Herlihy's small-object constructions, however, this array 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. An operation replaces only the blocks it modifies, and thus avoids copying the whole object. Before describing the code in Figure 5.1, we first explain how we provide the illusion of a contiguous array without resorting to copying it in its entirety.

Figure 5.2 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

```
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

shared vartype blktype = array[0..S − 1] of wordtype

shared var   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 (∀n : 0 ≤ n < B :: BANK[n] = N ∗ T + n ∧ BLK[N ∗ T + n] = (nth block of initial value))

private var  copy,                                                      /∗ Indices of p's copy blocks ∗/
             oldlst: array[0..T − 1] of 0..B + N ∗ T − 1;           /∗ Blocks to be reclaimed later ∗/
             ptrs: array[0..B − 1] of 0..B + N ∗ T − 1;   /∗ Pointers for p's logicial view of the object ∗/
             dirty: array[0..B − 1] of boolean;             /∗ Blocks to be replaced by p's operation ∗/
             dcnt: 0..T;                                          /∗ Number of blocks to be replaced ∗/
             i, blkidx: 0..B − 1;                    /∗ Counter and index of block accessed by Write ∗/
             v: wordtype;                                                /∗ Temporary value for Read ∗/
             ret: objrettype                                      /∗ Return value for p's operation ∗/

initially (∀n : 0 ≤ 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 7 else return v fi

procedure Write(addr: 0..B ∗ S − 1; val: wordtype)
3:  blkidx := addr div S;                                   /∗ Compute block index from address ∗/
    if ¬dirty[blkidx] then                                     /∗ Haven't changed this block before ∗/
4:     memcpy(BLK[copy[dcnt]], BLK[ptrs[blkidx]], sizeof(blktype));        /∗ Copy old block to new ∗/
5:     dirty[blkidx], oldlst[dcnt], ptrs[blkidx], dcnt := true, ptrs[blkidx], copy[dcnt], dcnt + 1
    fi;                                /∗ Install new block, record old block, prepare for next one ∗/
6:  BLK[ptrs[blkidx]][addr mod S] := val                                            /∗ Write new value ∗/

procedure LF_Op(op: optype; pars: paramtype)
    while true do                                               /∗ Loop until operation succeeds ∗/
7:     if Long_Weak_LL(BANK, ptrs) = N then                                  /∗ Load object pointer ∗/
          for i := 0 to B − 1 do dirty[i] := false od;                         /∗ No blocks copied yet ∗/
          dcnt := 0;
8:        ret := op(pars);                                             /∗ Perform operation on object ∗/
9:        if dcnt = 0 ∧ Long_VL(BANK) then return ret fi;                 /∗ Avoid unnecessary SC ∗/
10:       if Long_SC(BANK, ptrs) then                 /∗ Operation is successful, reclaim old blocks ∗/
             for i := 0 to dcnt − 1 do copy[i] := oldlst[i] od;
             return ret
       fi fi
    od
```

Figure 5.1: Lock-free implementation for a large object.

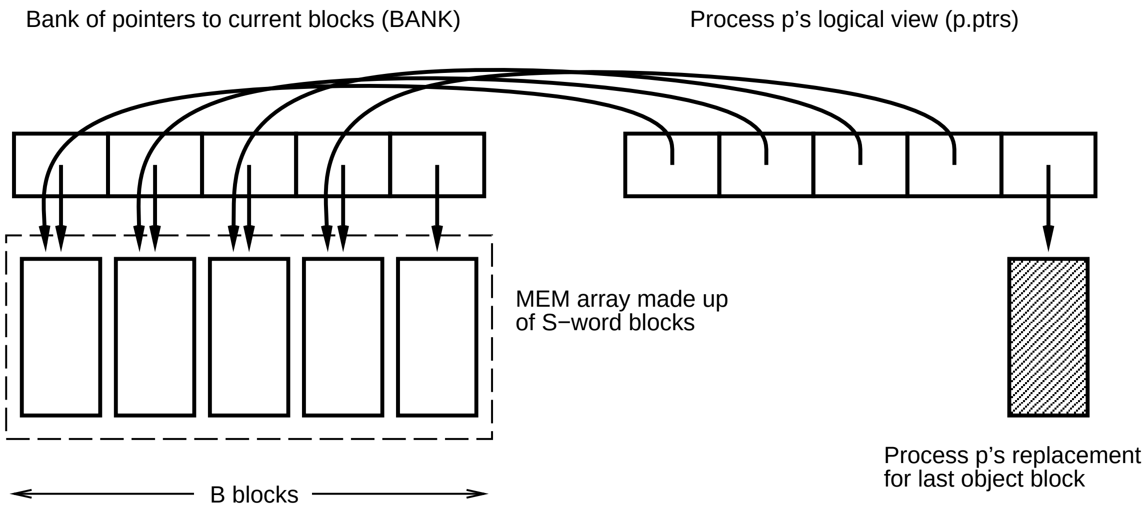


Figure 5.2: Implementation of the *MEM* array for large object constructions.

$MEM[2S-1]$ are stored in the second block, and so on. A bank of pointers, one to each block of the array, is maintained in order to record which blocks are currently part of the array. While performing an operation, a process p maintains a logical view of the current array. This logical view is represented by an array of pointers $p.ptrs$. At the beginning of the operation, $p.ptrs$ 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 the 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 current by writing the values in $p.ptrs$ to $BANK$. The $BANK$ array is read and modified by using the weak-LL and SC operations for large variables presented in Section 4.2.[1] In Figure 5.2, 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

[1]An extra parameter has been added to the procedures of Section 4.2 to explicitly indicate which shared variable is updated.

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 $MEM[x]$, the block that currently contains $MEM[x]$ must be identified. If p's operation modifies $MEM[x]$, 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. As a result, 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, consider Figure 5.3, which contains the actual code we used to implement a 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.

We now turn our attention to the code of Figure 5.1. In this figure, $BANK$ is a B-word shared variable, which is treated as an array of B pointers (actually indices into the BLK array), each of which points to a block of S words. Together, the B blocks pointed to by $BANK$ make up the implemented array MEM. We assume an upper bound T on the number of blocks modified by any operation. Therefore, in addition to the B blocks required for the array that stores the object, T "copy blocks" are needed for each process,

```
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 5.3: C code used for the enqueue operation of an array-based queue implementation. Comments show "usual" enqueue code.

giving a total of $B + NT$ blocks. These blocks are stored in the BLK array. As shown in Figure 5.1, blocks $BLK[NT]$ 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. 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.

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 line 10 succeeds. (Actually, read-only operations return from line 9 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 line 10, and therefore do not cause the SC operations of other processes to fail.) In each iteration, process p first reads *BANK* into *p.ptrs* using a B-word weak-LL. Recall from Section 4.2 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, and initializes the *dcnt* counter to zero. Process p uses the array *p.dirty* and the counter *p.dcnt* 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*. The *op* procedure performs the sequential operation by reading and writing the elements of the *BLKS* 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 *BLKS* currently represents the appropriate block of *MEM*, and modifies that block.) This reading and writing is performed by invoking the *Read* and *Write* procedures shown in Figure 5.1. 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. The *Write* procedure performs a write to a word of *MEM* by computing the index *blkidx* of the block containing the word to be written. If it has not already done so, the *Write* procedure then records that the block is "dirty" (i.e., has been modified) and copies the contents of the old block to one of p's copy blocks. Then, the copy block is linked into *p.ptrs*, making that block part of p's logical view of the *MEM* array, and the displaced old block is recorded in *oldlst* for possible reclaiming later. Finally, the appropriate word of the new block is modified to contain the value passed to the *Write* procedure.

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 subsection, 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.[2] 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 complexity (in terms of words) of the shared variables in the algorithm in Figure 5.1 is $\Theta(B+NTS)$, 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 $\Theta(NB+NTS)$. Because this algorithm is lock-free and not wait-free, the time for an operation to complete

[2] We model this behavior using a `goto` statement in our algorithms.

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 Chapter 4, 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, line 4 is executed at most T times during one operation, and the loop in line 10 executes at most T iterations. With these observations, the following theorem follows. (Recall that a full assertional proof is provided in Appendix A.1 for the wait-free version of this algorithm, which is presented in the next section.)

Theorem 5: 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[3] $\Theta(NB + NTS)$ and contention-free time complexity $\Theta(B + C + TS)$. □

These complexity figures compare favorably with those of Herlihy's lock-free construction. Consider the implementation of a queue. By storing head and tail "pointers" (actually, array indices, not pointers) in a designated block, an enqueue or dequeue 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 $\Theta(NB + NS)$, which should be small when compared to $\Theta(BS)$, the size of the queue. Contention-free time complexity

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

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., $\Omega(NBS)$. Also, contention-free time complexity is $\Omega(BS+C)$, since $\Omega(BS)$ 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 an operation implemented using our construction copies a part of the object only if the sequential version of the operation modifies that part of the object.

5.2 Wait-Free Universal Construction for Large Objects

Our wait-free construction for large objects is shown in Figures 5.4 through 5.6. The basic structure of this algorithm is similar to that of the lock-free construction presented in the previous subsection. 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

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 $(M \geq 2 * T)$

type $anctype$ = **record** op: $optype$; $pars$: $paramtype$; bit: 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 var 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 */
X: **record** pid: $0..N-1$; tag: 0..1 **end**; /* Current $BANK$ buffer */
BUF: **array**$[0..N-1, 0..1]$ **of** $banktype$; /* Buffers for $BANK$ */
$AuxObj$: **auxiliary array**$[0..B*S-1]$ **of** $wordtype$ /* Auxiliary value of object */

initially $X = (0,0) \ \wedge\ BUF[0,0].ret = N \ \wedge\ BUF[0][0].help = 0 \ \wedge\ AuxObj =$ initial object value $\wedge$
$(\forall p :: ANC[p].bit = 0 \ \wedge\ RET[N][p].applied = 0 \ \wedge\ RET[N][p].copied = 0\ \wedge$
$BUF[p][0].help = 0 \ \wedge\ BUF[p][1].help = 0) \ \wedge$
$(\forall n : 0 \leq n < B :: BUF[0,0].blks[n] = N*M+n\ \wedge$
$BLK[N*M+n] =$ (nth block of initial object value))

private var $copy$, /* Indices of p's copy blocks */
$oldlst$: **array**$[0..M-1]$ **of** $0..B+N*M-1$; /* Blocks to be reclaimed later */
$ptrs$: $banktype$; /* Process p's logicial view of the object */
$dirty$: **array**$[0..B-1]$ **of boolean**; /* Blocks to be replaced by p's operation */
$dcnt$: $0..M$; /* Number of blocks to be replaced */
rb, $oldrb$: $0..N$; /* Return block of process p and of previous object value */
$match$, a ,bit: 0..2; /* Counters for detecting outstanding and completed operations */
$applyop$: $optype$; $applypars$: $paramtype$; /* Operation to apply and its parameters */
rv: $objrettype$; /* Operation return value */
tmp, b: $0..N$; /* Witness of failed $Long\_Weak\_LL$, if any */
$done$, $loop$: **boolean**; /* Control variables */
i: $0..B-1$; $side$: 0..1; /* Variables for $Long\_Weak\_LL$ ... */
$curr$: **record** pid: $0..N-1$; tag: 0..1 **end**; /* ... and $Long\_SC$ */
try: $0..N-1$; /* Next process to help */
m: $0..M-1$; j, h: $0..N-1$; $k : 0..S-1$; /* Various counters */
$word$: $0..B*S-1$; val: $wordtype$; /* Variables for $Simulate\_Op$ */
$retval$: $objrettype$; $action$: $\{rd, wr, rt\}$;
$auxcopy$: **auxiliary array**$[0..B*S-1]$ **of** $wordtype$; /* Value of object during operations */
$hlplst$: **auxiliary list of** $tupletype$; /* List of helped operations */
$from$: **auxiliary** $\{29, 31, 40, 42, 48\}$ /* Used to model execution stack for proof */

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

Figure 5.4: Variable declarations for large object construction in Figures 5.5 and 5.6.

```
procedure Long_Weak_LL() returns 0..N
    curr, auxcopy := LL(X), AuxObj;
    for i := 0 to B − 1 do
        ptrs.blks[i] := BUF[curr.pid, curr.tag].blks[i]
    od;
    ptrs.help := BUF[curr.pid, curr.tag].help;
    ptrs.ret := BUF[curr.pid, curr.tag].ret;
    if VL(X) then return N
    else return X.pid fi

procedure Long_SC() returns boolean
    side := 1 − side;
    for i := 0 to B − 1 do
        BUF[p, side].blks[i] := ptrs.blks[i]
    od;
    BUF[p, side].help := ptrs.help;
    BUF[p, side].ret := ptrs.ret;
    if SC(X, (p, side)) then
        AuxObj := Apply_All(AuxObj, hlplst);
        return true
    else return false
    fi

procedure Read(addr: 0..B ∗ S − 1) returns wordtype
    v := BLK[ptrs.blks[addr div S]][addr mod S];
    if ¬VL(X) then goto 44 else return v fi

procedure Write(addr: 0..B ∗ S − 1; val: wordtype)
    blkidx := addr div S;                               /∗ Compute block index from address ∗/
    if ¬dirty[blkidx] then                              /∗ Haven't changed this block before ∗/
        for k := 0 to S − 1 do                          /∗ Copy old block to new ∗/
            tmpword := BLK[ptrs.blks[blkidx]][k];
            BLK[copy[dcnt]][k], k := tmpword, k + 1
        od;
        dirty[blkidx], oldlst[dcnt], tr[dcnt], ptrs.blks[blkidx], dcnt :=
                        true, ptrs.blks[blkidx], blkidx, copy[dcnt], dcnt + 1
    fi;         /∗ Mark block as changed, install new block, record old block, prepare for next one ∗/
    BLK[ptrs.blks[blkidx]][addr mod S], auxcopy[addr] := val, val       /∗ Write new value ∗/

procedure Simulate_Op(simop: optype; simpars: paramtype) returns objrettype

  while true do      /∗ Simulate_Op is not part of the construction; it merely facilitates the proof ∗/
      word, val, retval, action := select(0..B ∗ S − 1), select(wordtype), select(objrettype), select({rd, wr, rt});
      if action = rt then return retval
      elseif action = rd then val := Read(word)
      else Write(word, val)
      fi
  od

procedure Apply(pr: 0..N − 1)

    match := ANC[pr].bit;
    if RET[rb][pr].applied ≠ match then
        applyop := ANC[pr].op;
        applypars := ANC[pr].pars;
        rv := Simulate_Op(applyop, applypars);
        RET[rb][pr].val := rv;
        RET[rb][pr].applied := match;
        hlplst := hlplst · (pr, applyop, applypars, rv)
    fi

procedure Return_Block() returns 0..N

    tmp := Long_Weak_LL();
    if tmp ≠ N then
        return LAST[tmp]
    else
        return ptrs.ret
    fi
```

Figure 5.5: Wait-free large object construction.

```
procedure WF_Op(op: optype; pars: paramtype)
28: ANC[p], bit := (op, pars, (bit + 1) mod 3), (bit + 1) mod 3;          /* Announce operation */
29: from, done := 29, false; b := Return_Block();
30: while ¬done ∧ RET[b][p].copied ≠ bit do   /* Loop until update succeeds or operation is helped */
31:     from := 31;
        if Long_Weak_LL() = N then                                       /* Load object pointers */
32:         for i := 0 to B − 1 do dirty[i] := false od;  dcnt := 0;      /* No blocks modified yet */
            oldrb, ptrs.ret := ptrs.ret, rb;              /* Record old return block and install new one */
            for h := 0 to N − 1 do                                 /* Make private copy of return block */
33:             tmpval := RET[oldrb][h].val;
34:             RET[rb][h].val := tmpval;
35:             tmpbit := RET[oldrb][h].applied;
36:             RET[rb][h].applied := tmpbit
            od;
37:         if VL(X) then                                              /* Check if Long_SC will fail */
                for j := 0 to N − 1 do                                 /* Record applied operations */
38:                 a := RET[rb][j].applied;
39:                 RET[rb][j].copied := a
                od;
40:             hlplst, try, from := {}, p, 40; Apply(try);
41:             try, loop := ptrs.help, false;                              /* Apply own operation */
                while dcnt + T ≤ M ∧ ¬loop do          /* Help processes while sufficient space remains */
42:                 if try ≠ p then from := 42; Apply(try) fi;
43:                 try := (try + 1) mod N; if try = ptrs.help then loop := true fi
                od;
44:             LAST[p], ptrs.help := rb, try;                 /* Relay which return block was modified */
45:             if Long_SC() then                         /* Operation is successful, reclaim old blocks */
46:                 for m := 0 to dcnt − 1 do copy[m] := oldlst[m] od;
                    rb, done := oldrb, true
                fi
            fi
        fi;
47:     from := 47; b := Return_Block()                       /* Get current or recent return block */
    od;
48: from := 48; b := Return_Block();                              /* Not necessary for correctness, ... */
49: RET[b][p].copied := bit;                                           /* ... but simplifies the proof */
50: return RET[b][p].val                                               /* Get return value of operation */
```

Figure 5.6: Wait-free large object construction (continued from Figure 5.5).

a single operation.)

There are a number of differences between our lock-free and wait-free constructions. The principal difference is that processes in the wait-free construction "help" each other in order to ensure that each operation by each process is eventually completed. We explain the helping mechanism in detail later. There are also several differences between the presentations of these two algorithms that arise from the assertional correctness proof presented later. First, the algorithm in Figures 5.4 through 5.6 has several new auxiliary variables (*AuxObj*, *auxcopy*, *hlplst*, and *from*) and functions (*Apply_All*) that do not appear in the lock-free algorithm. These variables and functions are not actually implemented; they are provided solely for the purposes of the correctness proof, and may be ignored for now. Also, the algorithm in Figures 5.5 and 5.6 uses a special case of the the *Long_Weak_LL* and *Long_SC* implementations presented in Section 4.2. Here, we have eliminated the parameters passed to and from the *Long_Weak_LL* and *Long_SC* procedures, because all calls to these procedures use the same parameters. (A *Long_Weak_LL* operation by process p writes the current value of the *BANK*[4] variable into p's *ptrs* variable, and a successful *Long_SC* writes the current value of p's *ptrs* variable into the *BANK* variable.) Also, the *Long_Weak_LL* and *Long_SC* procedures in Figure 5.5 explicitly implement these operations on a variable of type *banktype*, rather than using the general large variable implementation presented in Figure 4.5. These simplifications eliminate the need to reason about parameter passing and type-casting mechanisms in the correctness proof. Finally, the *Simulate_Op* procedure is

[4]Observe that the *BANK* variable is not referred to explicitly in the code of Figures 5.5 and 5.6. The *BANK* variable is implemented by the *Long_Weak_LL* and *Long_SC* procedures using the X and *BUF* variables. The *Long_VL* operation is implemented directly as a VL operation on X (lines 13 and 37). By directly incorporating these implementations into the algorithm in Figures 5.5 and 5.6, we have eliminated the need to refer to *BANK* explicitly.

used to model the effects of a generic, user-supplied operation. The *Simulate_Op* procedure makes a nondeterministic sequence of calls to *Read* and *Write*, and then returns a value. This sequence of events is assumed to be the same as would be made by the implemented operation with the parameters given.

We now turn our attention to the helping mechanism used in our wait-free, large-object construction. The overall structure of our helping mechanism is the same as Herlihy's helping mechanism, described in Section 1.2. Specifically, helping is achieved by having each process p announce its operation in $ANC[p]$ (line 28 in Figure 5.6) before entering a loop (lines 30 through 47) that repeatedly attempts to perform its own operation (line 40) together with the operations of other processes (lines 41 through 43) until either p executes a successful SC (line 45) or p detects that its operation has been helped (line 30).

Despite the similarities in structure between our helping mechanism and Herlihy's, the details are quite different. 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. (Recall that *BANK* is implemented by the buffers in the *Long_Weak_LL* and *Long_SC* procedures in Figure 5.5, and is no longer referenced explicitly.) 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. The use of these two fields is described in detail below.

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 $RET[0]$ to $RET[N]$; 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 q, the return value of q's most recent operation, along with two 3-valued control fields: *applied* and *copied*. Together with $ANC[p].bit$, 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 $ANC[p].bit$ and the *applied* and *copied* fields of the current return block are all equal. When p announces a new operation (line 28), it also increments $ANC[p].bit$, thereby making it different from its *applied* and *copied* fields. As explained below, this indicates to other processes that p now has an outstanding operation.

A process q performs its own operation (line 40) and helps operations of other processes (line 42) by calling the *Apply* procedure, passing as a parameter the process pr to be helped. *Apply* checks whether process pr has an outstanding operation by comparing $ANC[pr].bit$ to the current *applied* field for process pr (lines 20 and 21). (More accurately, it compares $ANC[pr].bit$ to pr's *applied* field in a *copy* of the current return block. This copy is made by process q in lines 33 through 36. In the correctness proof in Appendix A.1, we show that, if the pr's *applied* field in this copy is different to that 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 pr's operation (lines 22 and 23), records the return value of the operation in its copy of the return block (line 24), and copies the value read from $ANC[pr].bit$ to pr's *applied* field in q's copy of the return block. Later, if q's SC is successful, then q's copy of the return block becomes current. Thus, pr's operation is not subsequently reapplied by another process, because that process finds that pr's *applied* field in the current return block equals $ANC[pr].bit$.

The *copied* field for process q in the current return block is used by q to detect when its operation has been completed. In lines 38 and 39, 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 $ANC[q].bit$ until the successful SC *after* the one that applies q's operation (described above). To see why two bits are needed to detect whether q's operation is complete, consider the scenario in Figure 5.7. In this figure, process p performs two operations. In the first, p's SC is successful, and p replaces $RET[5]$ with $RET[3]$ as the current return block at line 11. During p's first operation, q starts an operation. However, q starts this operation too late to be helped by p. Before p's execution of line 11, q determines that $RET[5]$ is the current return block (line 4). Now, p starts a second operation. Because p previously replaced $RET[5]$ as the current return block, $RET[5]$ is now p's private copy, so p's second operation uses $RET[5]$ to record the operations it helps. When p executes line 25, it changes q's *applied* bit to indicate that it has applied q's operation. Note that, at this stage, q's operation has only been applied to p's private object copy, and p has not yet performed its SC. However, if q reads the *applied* bit of $RET[5]$ (which it previously determined to be the current RET block) at line 4, then q incorrectly concludes that its operation has been applied to the object, and terminates prematurely.

It is similarly possible for q to detect that its *copied* bit in some return block $RET[b]$ equals $ANC[q].bit$ before the SC (if any) that makes $RET[b]$ current. However, because q's *copied* bit is updated only *after* its *applied* bit has been successfully installed as part of the current return block, it follows that some process must have previously applied

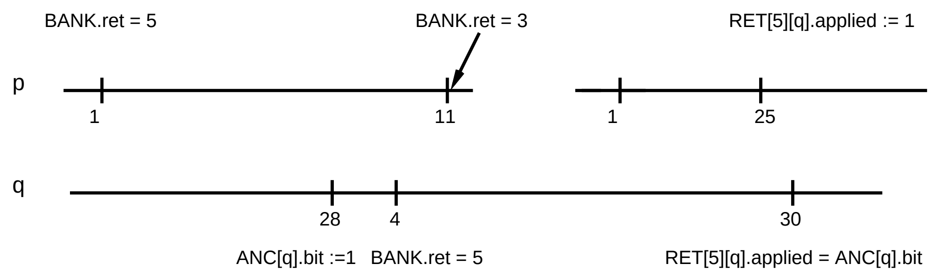


Figure 5.7: Process q prematurely detects that its *applied* bit equals $ANC[q].bit$.

q's operation. Thus, the use of the second bit (*copied*) ensures that q terminates correctly in this case (see line 30).

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 ID 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 $LAST[p]$ (line 44) before attempting to make that block current (line 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. 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 current return block, so its contents are still valid.

We now describe the use of the new *help* field of $BANK$. Recall that, 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. Therefore, in our construction, 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. 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 (lines 22 to 24). (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. 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, thereby guaranteeing wait-freedom. A full assertional proof is presented in Appendix A.1. This proof is quite long and tedious; we therefore present an overview of the proof in the next section.

5.3 Proof Overview for Algorithm in Figures 5.5 and 5.6

In this section we give an overview of the correctness proof for the algorithm in Figures 5.5 and 5.6. This overview is intended to aid understanding of the algorithm and to provide some intuition for the structure of the proof. It is *not* intended to be formal or rigorous. Relevant properties from the full proof that appears in Appendix A are indicated in parentheses.

In order to model the interaction of each process with our construction, we assume the existence of a statement 0 (not shown in Figures 5.5 and 5.6) that repeatedly calls *WF_Op* with appropriate parameters. When process p returns from the *WF_Op* procedure (line 50), p's program counter is set to 0 in preparation for p's next operation invocation. The *Simulate_Op* procedure (line 19) models user-supplied operations. This is achieved by using a nondeterministic choice to produce a sequence of read and write operations to the implemented BS-word array, followed by the return of a nondeterministic value. We use the *select* function, which takes a type parameter and returns an arbitrary value of that type, to achieve this nondeterminism.

The algorithm in Figures 5.5 and 5.6 is augmented with several auxiliary variables and functions, which are used to construct a total order over operation invocations that is consistent with the sequential semantics of the implemented object. The linearizability proof shows that each operation invocation is placed into this total order exactly once, that this occurs during the execution of that invocation (i.e., the constructed total order extends the partial order over invocations), and that each invocation returns the correct return value with respect to the total order constructed.

We now explain how the total order over operations is constructed. While performing operations, each process p maintains an auxiliary variable $p.auxcopy$ which records the changes made to p's local view of the implemented object. (Observe that, for each write operation invoked by an operation performed by process p, an equivalent write operation is performed on $p.auxcopy$ at line 18.) Process p records the operations it performs in an auxiliary list *p.hlplst* (line 25), which contains one tuple for each operation performed. Each tuple contains the identity of the process that invoked the operation, the operation itself, the parameters passed to that operation, and the value that would be returned by a sequential implementation of that operation, given the previous object state in *p.auxcopy*. Each time process p performs a successful SC operation (line 11), the operations in *p.hlplst* are linearized (i.e., added to the constructed total order) in the order that they appear in *p.hlplst*. This is recorded by updating the auxiliary variable *AuxObj*, which contains the current value of the implemented object. This is achieved by means of the *ApplyAll* function, which is defined below.

Definition: The *ApplyAll* function takes two parameters: a current object state (an array of BS words) and a list of tuples. *ApplyAll* scans the list in order and, for each tuple $(q, op, pars, ret)$, sequentially applies the operation *op*, with parameters *pars* to the current object state. *ApplyAll* returns the state of the object that results from all of these operation applications. (Note that this function is used only to facilitate the proof: it is not actually implemented.) □

Observe that, when p executes a LL at line 1, p also copies *Auxcopy* to *p.auxcopy*. Thus, because *Auxcopy* does not change between p's LL and p's successful SC (if it did change, then p's SC would fail), the new value written to *Auxcopy* when p performs its successful SC correctly reflects the sequential execution of the operations in *p.hlplst*. Thus, the return values of the operations in the total order constructed are consistent with the sequential semantics of those operations.

$$
\begin{array}{ll}
RV(p) & \equiv RET[BUF[X.pid, X.tag].ret][p].val \\
AV(p) & \equiv RET[BUF[X.pid, X.tag].ret][p].applied \\
CV(p) & \equiv RET[BUF[X.pid, X.tag].ret][p].copied \\
NORM(p) & \equiv AV(p) = ANC[p].bit \ \wedge \ CV(p) = ANC[p].bit \\
ST(p) & \equiv (AV(p)+1) \ \mathbf{mod} \ 3 = ANC[p].bit \ \wedge \ (CV(p)+1) \ \mathbf{mod} \ 3 = ANC[p].bit \\
APP(p) & \equiv AV(p) = ANC[p].bit \ \wedge \ (CV(p)+1) \ \mathbf{mod} \ 3 = ANC[p].bit
\end{array}
$$

Figure 5.8: Definitions used in the correctness proof for the algorithm in Figures 5.5 and 5.6.

We begin by introducing some of the definitions used in the proof, and by giving the intuition behind them. These definitions appear in Figure 5.8.

$RV(p)$, $AV(p)$, and $CV(p)$ are shorthand for the *val*, *applied*, and *copied* fields, respectively, for process p in the current return block (indicated by $BUF[X.pid, X.tag].ret$). The $NORM(p)$, $ST(p)$, and $APP(p)$ predicates represent the state of process p's current operation (if any). In the linearizability proof in Appendix A, we show that $NORM(p)$ holds while process p is at line 0 or line 28 — that is, while p is not executing an operation, or is just about to start one (I95). We also show that, during an operation by process p, p goes through three phases before returning. These are a *start* phase, during which $ST(p)$ holds, an *applied* phase, during which $APP(p)$ holds, and a *completed* phase, during which $NORM(p)$ holds again ((U2), (U3), (U4), and Claims 8, 9, and 10).

Process p starts an operation by incrementing $ANC[p].bit$ (line 28). Thus, because $NORM(p)$ holds before an operation by process p starts (I95), the execution of statement $p.28$ establishes $ST(p)$, thereby beginning the *start* phase of p's operation. The *applied* phase of p's operation begins when some process q performs a successful SC that changes the value of $AV(p)$. We show that this occurs only if process q performed p's operation, and hence copied $ANC[p].bit$ to p's *applied* field in q's local return block at lines 20 and 25 (I77). Thus, p's operation is linearized only at the transition from its *start* phase to its *applied* phase (i.e., the point at which $ST(p)$ is falsified and $APP(p)$ is established). This implies that p's operation is linearized exactly once, and that this occurs during the execution of p's invocation. Finally, we show that, immediately after p's operation is linearized, $RV(p)$ contains the correct return value for p's operation (I77), and that this remains true until p completes execution of its operation ((U4) and (U5)). We use this property to show that the value returned from line 50 by process p is the correct return value for its operation (I98).

The wait-freedom proof given in Appendix A shows that each of the three phases described above completes after a finite number of p's steps. In particular, because each process has sufficient copy space to help at least one other process with each operation, the value of the help counter ($BUF[X.pid, X.tag].help$) increases (modulo N) by at least one with each successful SC. Thus, if process p repeatedly fails to perform its own operation, then the help counter eventually ensures that the process that performs the next successful SC also performs p's operation. We also show that $AV(p) = ANC[p].bit$ is stable after that successful SC until the beginning of p's next operation ((U4) and (U5)). Thus, because each process copies the applied fields of its return block to the copied field (lines 38 and 39), the next successful SC (by p or by another process) establishes $CV(p) = ANC[p].bit$ (I77). We show that this assertion is stable until the beginning of p's next operation ((U4) and (U5)), and use this to show that the next time p tests the loop condition at line 30, that condition is false (I77). Thus, p's operation is guaranteed to complete execution. In Appendix A, we prove the following theorem.

Theorem 6: 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 $\Theta(N(NR + MS + B))$ and worst-case time complexity $\Theta(\lceil N/\min(N, \lfloor M/T \rfloor)\rceil (B + N(R + C) + MS))$.[5] □

5.4 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 5.9. 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

[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. Also, because $M \geq 2T$, $\lceil N/\min(N, \lfloor M/T \rfloor)\rceil$ is at most $N/2$.

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.

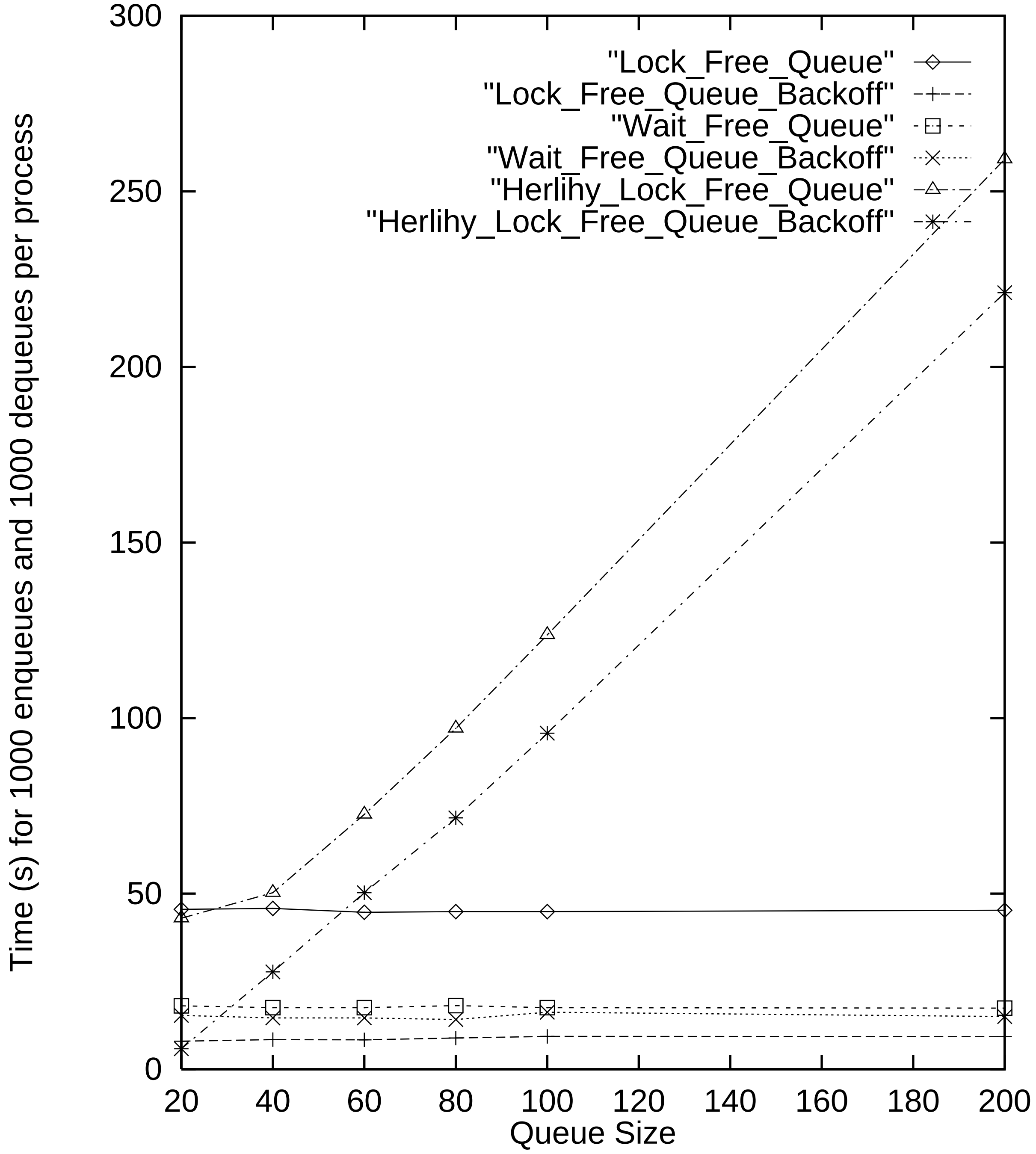


Figure 5.9: Comparison of our queue implementation to Herlihy's on a KSR multiprocessor.

In Herlihy's performance experiments on small objects [44], 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 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 to show the advantages of our constructions over Herlihy's. We also implemented a skew heap — the object considered by Herlihy in [44]. As a first step, 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 5.10. As this figure shows, our constructions performed about the same as they did when used to implement a queue. 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

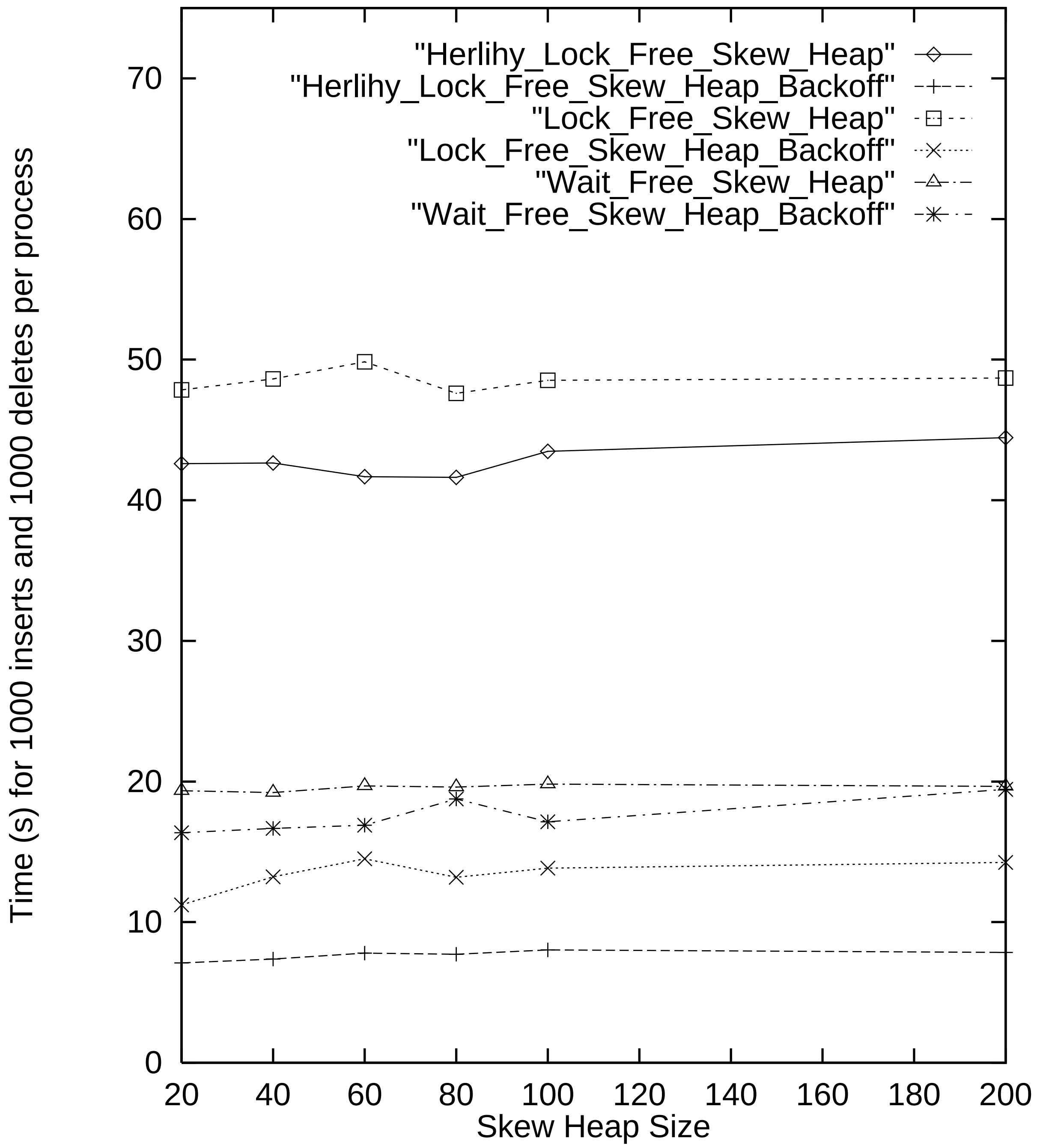


Figure 5.10: Comparison of our skew heap implementation to Herlihy's on a KSR multiprocessor.

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.

Chapter 6

Using k-Exclusion to Further Reduce Overhead

Most wait-free universal constructions — including the one presented in the previous chapter — have time complexity that depends on N, the total number of processes in the system. This is usually as a result of the need to tolerate $N-1$ simultaneous process delays or failures. However, in many applications, it is extremely unlikely that all N processes would simultaneously access the same shared object. Moreover, even if this scenario did arise, it would be unlikely for $N-1$ of them to fail or be delayed. Thus, wait-freedom is, in some sense, overkill in such applications. From a performance standpoint, it might pay to tolerate fewer simultaneous delays or failures, if that would incur lower overhead.

In this chapter, we present "tunable" object implementations, which allow the user to select a level of resiliency (that is, select how many delays or failures will be tolerated), and to adjust the associated overhead accordingly. In these implementations, a k-exclusion algorithm is used to protect access to a k-process, wait-free shared object implementation for some $k < N$. Provided at most k processes access the object concurrently, each process is guaranteed to complete each operation in a finite number of steps. On the other hand, if more than k processes access the object concurrently, then some are forced to wait by the k-exclusion algorithm.

We present several shared-memory algorithms for k-exclusion in which all process blocking is achieved through the use of "local-spin" busy waiting. As discussed in Section 2.1, algorithms that rely only on local spinning for process blocking reduce interconnect traffic, which is important for good performance. The algorithms we present are based on commonly-available synchronization primitives, are fast in the absence of contention, and exhibit scalable performance as contention rises. In contrast, all prior k-exclusion algorithms either require unrealistic atomic operations or perform badly.

The k-exclusion-based object implementations presented in this chapter are of interest because they combine the advantages of local-spin spin locks,[1] which perform well in the absence of process delays such as preemptions, and wait-free algorithms, which effectively tolerate such delays. We present performance results that show that, in multiprogrammed systems, these object implementations can perform much better than either wait-free or spin-lock-based object implementations. These results show that our k-exclusion algorithms are fast and scalable.

The *k-exclusion problem* was posed by Fischer et al. [34] as a generalization of the well-known mutual exclusion problem [29]. In the k-exclusion problem, the objective is to

[1]As mentioned in Chapter 2, a *spin lock* is a mutual exclusion algorithm that uses spinning to wait when another process holds the lock. A process *spins* by repeatedly testing a condition, for example by reading a shared variable, until that condition holds.

design a set of $N > k$ processes, each of which has a "critical section" of code. Each process can enter its critical section repeatedly, and at most k processes may be in their critical sections at any time. Progress must be guaranteed in the face of undetectable process halting failures. Specifically, if at most $k-1$ processes fail undetectably, then any nonfailed process that wants to enter its critical section eventually reaches it. In an important variant of the k-exclusion problem, originally posed by Attiya et al. [15], there is an added requirement that each process entering its critical section must obtain a unique "name" from a fixed set of k names. Following the terminology of Burns and Peterson [26], we call this variant of the k-exclusion problem the *k-assignment* problem.

In this chapter, we present several algorithms for solving the k-exclusion and k-assignment problems on shared-memory multiprocessors. We evaluate these algorithms in terms of their *remote-reference counts*, which specifies the number of remote accesses of shared memory required per critical section acquisition. An access is *remote* if it requires a traversal of the global interconnect between processors and shared memory, and *local* otherwise. Table 6.1 compares the algorithms of this chapter with previously published algorithms for k-exclusion and k-assignment. This table gives the remote-reference count of each listed algorithm, both under contention and in the absence of contention. (The contention-free remote-reference count is the worst-case number of remote memory accesses required for a critical section acquisition if no other process competes for that critical section. This notion is formalized in Section 6.1.) Table 6.1 also specifies the set of instructions used by each algorithm. Observe that all previously published algorithms have unbounded remote-reference counts under contention, and most have high remote-reference counts even

Ref.	Remote-Reference Complexity With Contention	Remote-Reference Complexity Contention-Free	Instructions Used
[34]	∞	$\Theta(1)$	Large Critical Sections
[35]	∞	$\Theta(1)$	Large Critical Sections
[30]	∞	$\Theta(N^2)$	Safe Bits
[3]	∞	$\Theta(N)$	Atomic Read and Write
[26]	∞	$\Theta(N)$	Atomic Read and Write
Thm. 9	$\Theta(k \log(N/k))$	$\Theta(1)$	Fetch-and-Add, Test-and-Set
Thm. 10	$\Theta(c)$	$\Theta(1)$	Fetch-and-Add, Test-and-Set
Thm. 13	$\Theta(k \log(N/k))$	$\Theta(1)$	Fetch-and-Add, Test-and-Set
Thm. 14	$\Theta(c)$	$\Theta(1)$	Fetch-and-Add, Test-and-Set
Thm. 17	$\Theta(k \log(N/k))$	$\Theta(1)$	Above and Compare-and-Swap
Thm. 18	$\Theta(c)$	$\Theta(1)$	Above and Compare-and-Swap

Table 6.1: A comparison of N-process k-exclusion algorithms for shared-memory systems. In the first column of time complexity figures, c is the level of contention. For the algorithms of Theorems 9 and 10, time complexity under contention is as stated only for cache-coherent machines. The compare-and-swap-based algorithms of Theorems 17 and Theorem 18 improve upon the algorithms of Theorems 13 and 14 by having lower space complexity. The algorithms of Theorem 9 through Theorem 18 all use atomic reads and writes in addition to the instructions listed.

in the absence of contention. In addition, the algorithms of [34] and [35] assume the existence of large mutually exclusive critical sections that are executed atomically.

Our decision to evaluate our k-exclusion algorithms by their remote-reference counts — that is, to distinguish between local and remote accesses of shared memory — is motivated by recent work on local-spin spin locks [5, 12, 38, 72, 97, 98]. In such locks, the impact of the processor-to-memory bottleneck is minimized by structuring programs so that processes busy wait only on locally-accessible shared variables. In practice, a shared variable can be made locally-accessible by storing it in a local cache line or in a local partition of distributed shared memory. Performance studies presented in [12, 38, 72, 97, 98] show that minimizing remote memory accesses is important for scalable performance in the design of synchronization algorithms.

Although the k-assignment problem may seem to be much harder than the k-exclusion problem, we show that if one allows reasonable synchronization primitives, then any k-exclusion algorithm can easily be extended to solve the k-assignment problem. Therefore, this chapter is almost entirely devoted to algorithms for k-exclusion. Our conversion of k-exclusion algorithms to k-assignment algorithms involves using a simple *long-lived renaming* algorithm that allows each process to acquire a name before entering its critical section, and to release that name upon exiting its critical section. The renaming algorithm we present is based on test-and-set, and has time complexity that is directly proportional to contention. A generalization of this algorithm is presented in Chapter 7 and proved correct in Appendix C.2.

The k-exclusion algorithms we present employ only local spins for process blocking. The first few algorithms we present are designed for implementation on cache-coherent machines. In these algorithms, spins are local only if there is an underlying cache-coherence mechanism. The remaining algorithms in this chapter do not require cache coherence. Hence, they can be implemented on distributed shared-memory machines that do not have coherent caches. For both classes of machines, we present algorithms that have $\Theta(k \log(N/k))$ remote-reference counts under contention and algorithms that have remote-reference counts that are directly proportional to contention (see Table 6.1). As shown in Table 6.1, all of these algorithms have constant remote-reference counts in the absence of contention, and are based on commonly-available synchronization primitives.

Most of the claims made above concerning scalable performance are based on remote-reference counts. In the latter part of the chapter, we put these claims to the test by evaluating the performance of our algorithms within the context of a particular application. The application we consider is that of implementing shared objects in systems that are multiprogrammed. In our experiments, we compare wait-free and spin-lock-based object implementations to object implementations that incorporate both wait-free and lock-based techniques. These implementations are $(k-1)$-*resilient*, which means that they can withstand undetectable halting failures of up to $k-1$ processes. As explained in Section 6.5, wait-free objects actually are a special case of this definition: an N-process object implementation is *wait-free* if and only if it is $(N-1)$-resilient.

The $(k-1)$-resilient shared object implementations we consider are obtained by encasing a wait-free, k-process object implementation within a k-assignment "wrapper". This wrapper permits only k processes to access the wait-free implementation concurrently, and assigns these processes unique names from a range of size k to use within that implementation. This approach allows $k-1$ process halting failures to be tolerated. From the object designer's standpoint, k is a parameter that determines the degree to which halting failures can be tolerated. Performance can be optimized by "tuning" this parameter.

The performance experiments we present show that, for both cache-coherent and distributed shared-memory multiprocessors, for suitable choices of k, $(k-1)$-resilient objects implemented using our algorithms are faster and scale better than objects implemented using either wait-free or lock-based algorithms. These results validate our claims that our k-exclusion algorithms are both fast and scalable. To our knowledge, the experiments we present are the first to demonstrate the advantages that resilient object implementations have over lock-based implementations in multiprogrammed systems. These advantages do not seem to be widely appreciated,[2] despite the considerable attention resilient objects have received in the literature.

The remainder of this chapter is organized as follows. In Section 6.1, we present definitions and notation that will be used in the rest of the chapter. In Section 6.2, we dispense with the k-assignment problem as discussed above by showing that a simple renaming[3] algorithm can be combined with any k-exclusion algorithm to solve k-assignment. We then present k-exclusion algorithms for cache-coherent and distributed shared-memory machines in Sections 6.3 and 6.4, respectively. We follow these sections with performance results in Section 6.5. Correctness proofs for the distributed shared-memory algorithms appear in Appendix B.

6.1 Preliminaries

A program that solves the *k-exclusion* problem consists of $N > k$ processes, which are numbered from 0 to $N - 1$. Each process begins execution in a *noncritical section*, and cycles through its noncritical section, an *entry section*, a *critical section*, and an *exit section*. Unless stated otherwise, we assume that no variable (other than program counters) appearing in any entry or exit section is modified in any noncritical or critical section.

A program that solves the k-exclusion problem must be able to cope with process halting failures. In keeping with the definitions given in Chapter 3, a process p is *faulty* in a history $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$ if and only if for some $i \geq 0$, process p is outside of its noncritical section

[2] All previously published performance evaluations of resilient objects that we know of assume a one-process-per-processor model of computation [10, 44]. Discussions with researchers in this area have led us to believe that many researchers are not aware of the performance benefits of resilient objects under multiprogramming.

[3] The renaming problem is studied in greater detail in the next chapter.

at state t_i, and $t_i \overset{s_i}{\rightarrow} t_{i+1} \overset{s_{i+1}}{\rightarrow} \cdots$ includes no statement executions of process p. Informally, a nonfaulty process can halt only in its noncritical section. Note that this implies that a nonfaulty process cannot halt in its critical section.

We now state the key safety and progress requirements for the k-exclusion problem. Let $ES(p)$ be a state assertion that is true if and only if the value of process p's program counter equals a label of a statement appearing in its entry section. Similarly, let $CS(p)$ be a state assertion that is true if and only if the value of process p's program counter equals a label of a statement appearing in its critical section. Then, a program that solves the k-exclusion problem must satisfy the following properties.

- *k-Exclusion*: $|\{p : 0 \leq p < N :: CS(p)\}| \leq k$ is an invariant. Informally, at most k processes can execute their critical sections at the same time.

- *Starvation-Freedom*: For each process p, $ES(p)$ *leads-to* $CS(p)$ in each history in which p is nonfaulty and at most $k - 1$ processes are faulty. Informally, if a nonfaulty process is in its entry section, then that process eventually executes its critical section, provided that fewer than k processes have failed. (We also require that each process in its exit section eventually enters its noncritical section; this requirement holds trivially in all of our k-exclusion algorithms, so we do not consider it further.)

The *k-assignment* problem extends the k-exclusion problem by requiring each process p to have a private variable $p.name$ ranging over $\{0, \ldots, k-1\}$. If distinct processes p and q are in their critical sections, then it is required that $p.name \neq q.name$. In other words, we require that $(\forall p, q : p \neq q :: (CS(p) \ \wedge \ CS(q)) \Rightarrow (p.name \neq q.name))$ is an invariant.

As mentioned previously, we focus on cache-coherent and distributed shared-memory machines, and evaluate our algorithms by counting "remote" references of shared memory. On distributed shared-memory machines, each shared variable is local to one processor, and remote to all others. Thus, the distinction between local and remote memory references is straightforward. On cache-coherent machines, making this distinction is more problematic. The main difficulty is in determining how many cache misses a busy-waiting loop generates. In our cache-coherent algorithms, all busy waiting is by means of simple loops of the form "**while** $Q = p$ **do od**", where Q is a shared variable and p is the process identifier of the spinning process. We assume that such a loop generates at most two remote memory references. In particular, we assume that the first read of Q generates a remote memory reference that causes a copy of Q to migrate to p's local cache. Subsequent reads before Q is written are therefore local. When another process modifies Q, the cache entry is invalidated, so the next read of Q generates a second remote memory reference. In our cache-coherent algorithms, each process modifies Q only by assigning its own process identifier to Q, so the loop terminates after this second remote read. These assumptions correspond to an idealized write-invalidate cache-coherence protocol, where a cached copy of Q is invalidated only by writes to Q. In fact, other invalidations are possible, for example because of preemption. However, these other invalidations should be relatively rare, so our model closely reflects reality.

We evaluate the algorithms in this chapter by counting (under the assumptions of the previous paragraph) the worst-case number of remote memory references required for any process to enter and then exit its critical section. We say that a k-exclusion or

k-assignment algorithm has *remote-reference count* R if and only if each matching entry and exit section of any process of that algorithm together generate at most R remote memory references. For some algorithms, the remote-reference count depends on the level of contention. For any state of any history, we define *contention* at that state to be the number of processes outside their noncritical sections at that state. We say that *contention is at most* c *in a history* if and only if contention is at most c at each state of that history. We say that a k-exclusion or k-assignment algorithm has *remote-reference count* R *if contention is at most* c if and only if each matching entry and exit section of any process of that program together generate at most R remote memory references in any history for which contention is at most c. When we refer to the remote-reference count of a program in the *absence of contention*, we mean its remote-reference count if contention is exactly one. Algorithms that do not employ local spinning can generate an unbounded number of remote memory references. We therefore say that a k-exclusion algorithm is a *local-spin* k*-exclusion algorithm* if it has a bounded remote-reference count. In addition to the conventions described in Chapter 3, we use the following in the remainder of this chapter.

Notational Conventions: In addition to the notation given in Chapter 3, in this chapter, we sometimes label sequences of statements such as noncritical and critical sections, and the *Acquire* and *Release* procedures introduced later in this chapter. This allows us to reason about steps of these sequences of statements. If s labels a sequence of statements, then $p@s$ holds if and only if some statement within that sequence is enabled for execution. Also, for brevity, we refer to k-exclusion for N processes as (N, k)*-exclusion*; similarly for (N, k)*-assignment*. $\square$

```
shared variable
    X : array[0..k-1] of boolean
initially
    (∀i : 0 ≤ i ≤ k-1 :: ¬X[i])

private variable
    name : 0..k-1

while true do
0:  Noncritical Section;
1:  Acquire(N,k);                                          /* Entry section for (N,k)-exclusion */
2:  name := 0;
3:  while test_and_set(X[name]) do name := name + 1 od;    /* Set first clear bit ... */
4:  Critical Section using name name;                      /* ... to get a name */
5:  X[name] := false fi;                                   /* Release name by resetting bit found */
6:  Release(N,k)                                           /* Exit section for (N,k)-exclusion */
od
```

Figure 6.1: Algorithm for k-assignment using test-and-set for renaming.

6.2 k-Assignment

As explained in the introduction, the k-assignment problem can be solved by combining a solution to the long-lived renaming problem with a program that solves the k-exclusion problem. The long-lived renaming problem, in which processes repeatedly acquire and release unique names from a fixed name space, is studied in detail in Chapter 7.

Figure 6.1 depicts a program that solves the k-assignment problem in the manner described above. The entry and exit sections of the k-exclusion algorithm being used are denoted by $Acquire(N, k)$ and $Release(N, k)$, respectively. The renaming mechanism employs a sequence of test-and-set bits, one per name. In order to obtain a name, a process tests each bit in order, until a test-and-set succeeds (line 3). The bit $X[j]$ is associated with name j, where $0 \leq j < k$. A process that has obtained name j releases it by simply clearing $X[j]$ (line 5).

A generalization of the renaming algorithm employed in Figure 6.1 is presented

```
shared variable
    X : (k − N)..k;
    Q : queue of 0..N − 1
initially
    X = k ∧ Q = null

process p                                                    /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  ⟨ if fetch_and_add(X, −1) ≤ 0 then        /* If no critical section slots are available... */
        Enqueue(p, Q) ⟩;                                 /* ... then get into queue ... */
2:      while Element(p, Q) do /* null */ od         /* ... and busy wait until released */
    fi;
3:  Critical Section;
4:  ⟨ Dequeue(Q);                                      /* Remove first process from Q */
    fetch_and_add(X, 1) ⟩                   /* Increase counter of available slots again */
od
```

Figure 6.2: (N, k)-exclusion using atomic queue procedures.

in more detail in Chapter 7 and proved correct in Appendix C.2. As the correctness proof presented there shows, if a process is about to perform a test-and-set operation on $X[i]$, then $\neg X[j]$ holds for some j where $i \leq j < k$. Thus, if a process has unsuccessfully tested bits $X[0]$ through $X[k-2]$, then $\neg X[k-1]$ holds, so the kth test-and-set will succeed. Therefore, although the loop at line 3 of Figure 6.1 has no explicit bound, termination is guaranteed after at most k iterations. Note that this renaming algorithm has remote-reference count $k+1$. Also, if contention is at most c, then it has remote-reference count $c+1$. Because each process has a private *name* variable, the renaming algorithm requires $\Theta(N)$ space. Thus, we have the following theorem.

Lemma 1: Suppose that $Acquire(N, k)$ and $Release(N, k)$ can be implemented with remote-reference count B, with remote-reference count C if contention is at most c, and with space complexity $\Theta(D)$. Then, (N, k)-assignment can be implemented with remote-reference count $B + k + 1$, with remote-reference count $C + c + 1$ if contention is at most c, and with space complexity $\Theta(D + N)$. $\square$

6.3 k-Exclusion on Cache-Coherent Machines

In this section, we present several fast k-exclusion algorithms for cache-coherent machines. We begin by explaining the key insight on which all of our k-exclusion algorithms are based.

On first thought, it may seem that the k-exclusion problem could be efficiently solved by simply modifying a queue-based spin lock [12, 38, 72] so that a process waits in the queue only if k other processes are already in their critical sections. Before giving our first algorithm, we explain why this simple approach is problematic. Consider the simple (unrealistic) queue-based (N, k)-exclusion algorithm in Figure 6.2. The shared variable X in this algorithm counts the number of processes that may safely enter the critical section. X is initially k. When $X \leq 0$, a process trying to enter the critical section waits in the queue Q. $Enqueue(p, Q)$ and $Dequeue(p, Q)$ are the normal queue operations, and $Element(p, Q)$ is a function that returns true if and only if p is in Q. Multi-line atomic statements are enclosed in angle brackets.

Aside from the multi-line atomic statements, there are two difficulties involved with implementing this algorithm. First, the queue operations typically require several atomic steps if implemented using only simple primitives. Such an implementation is complicated by the possibility that a process may fail after having only partially executed a queue operation. Second, a queue imposes a linear order on the waiting processes. If a process in the queue fails, then other processes in the queue are blocked.

However, both problems disappear when $N = k + 1$, because at most one process ever waits in the queue in this case, which allows us to implement the queue with one atomic instruction. This insight is the basis of the algorithms we present. Specifically, we concentrate on solving $(k+1, k)$-exclusion, and then inductively apply such a solution to solve (N, k)-exclusion.

Our (N, k)-exclusion algorithm for cache-coherent machines is shown in Figure 6.3. The *Acquire* and *Release* procedures in this figure are inductively assumed to implement $(N, k+1)$-exclusion. Thus, as is stated formally below, at most $k+1$ processes concurrently execute statements 2 through 9. In this algorithm, the idea of having one process in the queue is approximated by using a shared variable Q to store the identifier of the process that is "in the queue". A process can perform the dual functions of enqueueing itself and dequeueing the previously-queued process by simply assigning its own process identifier to Q. The variable X in Figure 6.3 is used in the same way as in the queue-based algorithm of Figure 6.2. As mentioned above, the *Acquire* and *Release* procedures in Figure 6.3 are inductively assumed to implement $(N, k+1)$-exclusion. That is, we assume the following properties, where the latter two are required to hold only if process p is nonfaulty and at most $k-1$ processes are faulty.

invariant $|\{q :: q@\{2..8\}\}| \leq k+1$ (I1)

$p@1$ *leads-to* $p@2$ (L1)

$p@9$ *leads-to* $p@0$ (L2)

It is assumed that the variables used by $Acquire(N, k+1)$ and $Release(N, k+1)$ are distinct from those in the remainder of the algorithm. Note that if $N = k+1$, then $Acquire(N, k+1)$ and $Release(N, k+1)$ are trivially implemented by skip statements. We later use this as the basis of an induction to show that (N, k)-exclusion can be implemented

```
shared variable
     X : −1..k;                                         /* Counter of available slots */
     Q : 0..N − 1                                       /* Spin location */
initially
     X = k

process p                                               /* 0 ≤ p < N */
while true do
0:   Noncritical Section;
1:   Acquire(N,k + 1);                                  /* Entry section of (N,k + 1)-exclusion */
2:   if fetch_and_add(X, −1) = 0 then                   /* No slots available */
3:        Q := p;                                       /* Initialize spin location */
4:        if X < 0 then                                 /* Still no slots available - must wait */
5:             while Q = p do /* null */ od             /* Busy-wait until released */
     fi   fi;
6:   Critical Section;
7:   fetch_and_add(X, 1);                               /* Release a slot */
8:   Q := p;                                            /* Release waiting process (if any) */
9:   Release(N,k + 1)                                   /* Exit section of (N,k + 1)-exclusion */
od
```

Figure 6.3: (N, k)-exclusion on a cache-coherent machine

efficiently.

The algorithm shown in Figure 6.3 is proved correct by establishing the following properties.

- *k-Exclusion*: **invariant** $|\{p :: p@6\}| \le k$
- *Starvation-Freedom*: If process p is nonfaulty and at most $k - 1$ processes are faulty, then $p@1$ *leads-to* $p@6$. (Given (L2), Starvation-Freedom for the exit section is trivial.)

Several properties are presented below in order to prove k-Exclusion and Starvation-Freedom. The first two of these properties are straightforward to prove directly from the program text, and are therefore stated without proof.

invariant $X = k - |\{p :: p@\{3..7\}\}|$ (I2)

invariant $X < 0 \Rightarrow (\exists p :: p@3 \;\vee\; (p@\{4, 5\} \;\wedge\; Q = p))$ (I3)

The invariant given next establishes the k-Exclusion property.

invariant $|\{p :: p@6\}| \leq k$ (I4)

Proof: If $X \geq 0$, then by (I2), $|\{p :: p@\{3..7\}\}| \leq k$ holds, so (I4) holds. If $X < 0$, then by (I3), $(\exists p :: p@\{3, 4, 5\})$ holds, so by (I1), (I4) holds. □

The following simple *unless* property, which follows immediately from the program text, is used in the proof of Starvation-Freedom.

$p@5 \ \wedge \ Q \neq p \quad \textit{unless} \quad p@6$ (U1)

Starvation-Freedom: If process p is nonfaulty and at most $k - 1$ processes are faulty, then $p@1$ *leads-to* $p@6$.

Proof: By (L1) and (L2), the only risk to Starvation-Freedom is that a nonfaulty process p is blocked forever at statement $p.5$. Process p only reaches $p.5$ by executing $p.4$ when $X < 0$ holds. By (I2), this implies that $|\{p :: p@\{3..7\}\}| > k$ holds when $p.4$ is executed. By the assumption that at most $k - 1$ processes are faulty, this implies that there is a nonfaulty process $q \neq p$ such that $q@\{3..7\}$ holds when $p.4$ is executed.

If $p@5 \ \wedge \ Q = p$ holds after $p.4$ is executed, then process q is not blocked at $q.5$ because $q \neq p$. If $p@5 \ \wedge \ Q = p$ continues to hold, then q, being nonfaulty, eventually executes $q.8$ and establishes $Q = q$, and hence $Q \neq p$. Thus, $p@5 \ \wedge \ Q \neq p$ holds at some state after $p.4$ is executed. Therefore, by (U1), $p@6$ eventually holds, because p is nonfaulty. This concludes the proof of Starvation-Freedom. □

For $N = k+1$, $\textit{Acquire}(N, k+1)$ and $\textit{Release}(N, k+1)$ can be trivially implemented by skip statements. Thus, by the above properties, the algorithm shown in Figure 6.3 can be used to implement $(k+1, k)$-exclusion with remote-reference count 7 (recall that the loop at statement 5 is assumed to generate at most two remote memory references). Using this result, we can inductively solve (N, k)-exclusion. The inductive algorithm consists of $N - k$ nested "levels", where each level corresponds to an instance of the algorithm of Figure 6.3. The outermost level solves $(N, N-1)$-exclusion, the next level solves $(N-1, N-2)$-exclusion, and so on. For this and other inductive solutions considered in this section to be correct, we must insist that different instances of the algorithm of Figure 6.3 use distinct Q and X variables. (This point may seem obvious, but we violate it later in Section 6.4.2.)

Since each level has remote-reference count 7, the algorithm described in the previous paragraph has remote-reference count $7(N-k)$. Each level requires constant space, so the algorithm's space complexity is $\Theta(N)$. These observations give us the following result. (In this theorem and those that follow, we only list atomic operations other than reads and writes.)

Theorem 7: Using fetch-and-add, (N, k)-exclusion can be implemented on a cache-coherent machine with remote-reference count $7(N - k)$ and space complexity $\Theta(N)$. $\square$

This inductive algorithm requires $\Theta(N)$ remote memory references, which is a significant disadvantage. Note, however, that Theorem 7 implies that $(2k, k)$-exclusion can be implemented with remote-reference count $7k$. We can use such an algorithm inductively as a "building block" to obtain a more efficient implementation of (N, k)-exclusion. Specifically, we can achieve logarithmic remote-reference count by arranging these building blocks in a tree that halves the number of process at each level, until only k remain. Figure 6.4 depicts this approach for $8k$ processes. This algorithm is shown in Figure 6.5. In this figure, it is inductively assumed that *Acquire_left* and *Release_left* correctly implement $(\lceil N/2 \rceil, k)$-

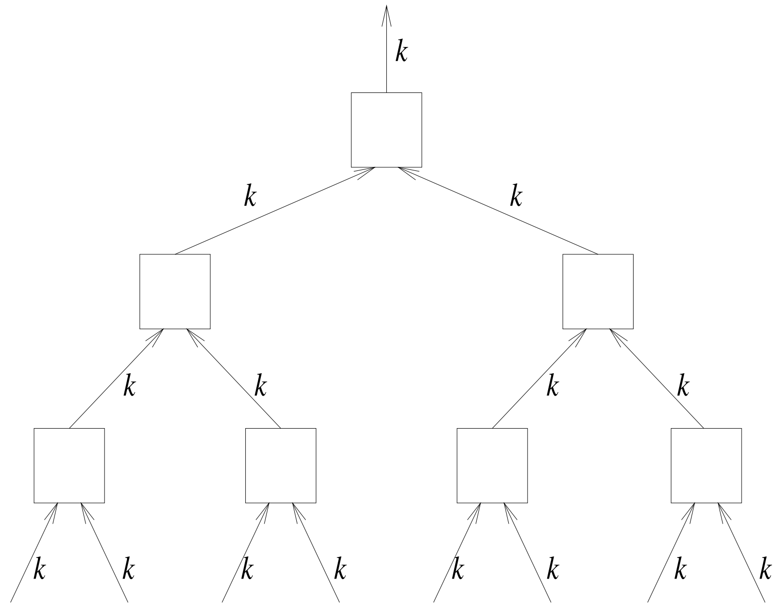


Figure 6.4: Implementing *Acquire*$(8k, k)$ in a tree. *Release*(N, k) is implemented analogously. Each arrow represents a set of processes. Solid boxes represent *Acquire*$(2k, k)$.

exclusion, and *Acquire_right* and *Release_right* correctly implement $(\lfloor N/2 \rfloor, k)$-exclusion. In addition, *Acquire_middle* and *Release_middle* are assumed to correctly implement $(2k, k)$-exclusion using the algorithm given in Figure 6.3 with $N = 2k$.

Before continuing, we should point out a key property of our $(k + 1, k)$-exclusion algorithm that allows it to be efficiently used in inductive applications as described above: this algorithm does not require a process in its entry section to know the identity of any other process in advance. To see this, note that in the proof of the algorithm of Figure 6.3, it does not matter how *Acquire*$(N, k+1)$ and *Release*$(N, k+1)$ are actually implemented — they could be implemented using the tree algorithm of the previous paragraph, or any other algorithm that is a correct $(N, k+1)$-exclusion algorithm. The correctness of the algorithm of Figure 6.3 also does not depend on which set of up to $k + 1$ processes actually make it past *Acquire*$(N, k + 1)$ to execute the code given for $(k + 1, k)$-exclusion. Without this

```
process p                                              /* 0 ≤ p < N */
while true do
    Noncritical Section;
    if p < ⌈N/2⌉ then                                  /* Half access left subtree */
        Acquire_left(⌈N/2⌉, k)
    else                                               /* Half access right subtree */
        Acquire_right(⌊N/2⌋, k)
    fi;
    Acquire_middle(2k, k);                             /* All access root of tree */
    Critical Section;
    Release_middle(2k, k);
    if p < ⌈N/2⌉ then                                  /* Half access left subtree */
        Release_left(⌈N/2⌉, k)
    else                                               /* Half access right subtree */
        Release_right(⌊N/2⌋, k)
    fi;
od
```

Figure 6.5: (N, k)-exclusion in a tree.

property, it might be difficult to efficiently apply such an algorithm inductively as done here. In particular, loops with $\Theta(N)$ time complexity might be required to detect the identity of competing processes at each instance of the inductively-applied algorithm, resulting in performance that does not scale.

The algorithm in Figure 6.5 can be proved correct by an induction on the tree depth, using the fact that our $(2k, k)$-exclusion algorithm is correct. The depth of the tree is $\lceil \log_2(N/k) \rceil$, and the remote-reference count of accessing each $(2k, k)$-exclusion building block is $7k$. The algorithm uses $2\lceil N/k \rceil - 1$ $(2k, k)$-exclusion building blocks in total, and by Theorem 7, each build block requires $\Theta(k)$ space. Thus, the algorithm shown in Figure 6.5 yields the following result.

Theorem 8: Using fetch-and-add, (N, k)-exclusion can be implemented on a cache-coherent machine with remote-reference count $7k\lceil \log_2(N/k) \rceil$ and space complexity $\Theta(N)$. □

The tree approach offers a significant improvement over the approach used in Theorem 7. However, we would like to further reduce the number of remote memory

```
shared variable
    X : boolean;                                    /* Test for fast path */
initially
    X = false

process p                                           /* 0 ≤ p < N */
private variable
    slow : boolean                                  /* Path taken */

while true do
    Noncritical Section;
    slow := test_and_set(X);                        /* Try to get fast path */
    if slow then                                    /* Take slow path */
        Acquire(N − 1, k)                           /* Slow path */
    fi;
    Acquire(k + 1, k);                              /* Fast path */
    Critical Section;
    Release(k + 1, k);
    if slow then                                    /* Check if slow path was taken */
        Release(N − 1, k)
    else
        X := false                                  /* Release fast path */
    fi
od
```

Figure 6.6: (N, k)-exclusion with a "fast path".

references performed when contention is low. This can be achieved by adding a "fast path", as shown in Figure 6.6. A test-and-set instruction is used to select one process that directly executes *Acquire*$(k+1, k)$. The remaining $N-1$ processes must first execute *Acquire*$(N-1, k)$, thereby ensuring that at most $k+1$ processes concurrently access the innermost $(k+1, k)$-exclusion algorithm. This approach is depicted in Figure 6.7, in which the dotted box represents *Acquire*$(N-1, k)$. Using this algorithm, if contention is one, then the test-and-set of the single competing process succeeds. Hence, that process executes only the innermost *Acquire*$(k+1, k)$ and *Release*$(k+1, k)$. Observe that this process performs at most 9 remote memory references: 2 are required to set and clear the test-and-set bit, and at most 7 are required for the $(k+1, k)$-exclusion.

The performance under contention for the algorithm shown in Figure 6.6 is determined by the implementation of $(N-1, k)$-exclusion — the "slow path". One alternative

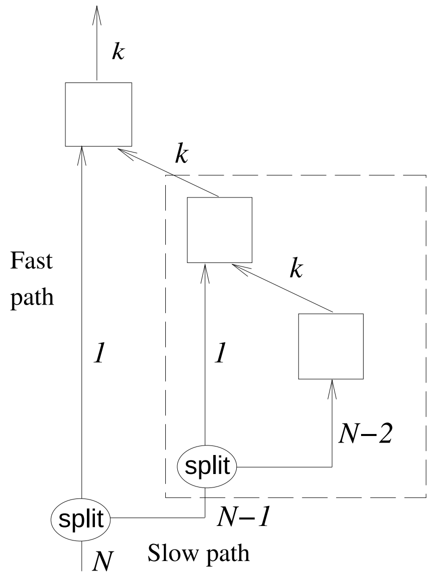


Figure 6.7: Implementing *Acquire*(N, k) using fast paths. Each arrow represents a set of processes. The split is implemented using test-and-set, and causes a process that executes alone to take the fast path. Solid boxes represent *Acquire*$(k+1, k)$. The dotted box represents *Acquire*$(N-1, k)$. One approach for implementing *Acquire*$(N-1, k)$ using nested fast paths is depicted for $N = k+3$ (so $N-2 = k+1$).

is to use a tree approach like the one illustrated in Figure 6.4. In this case, a process performs at most $7k\lceil \log_2(N/k)\rceil + 8$ remote memory references: 1 is required for an unsuccessful test-and-set, at most 7 are required for the innermost $(k+1, k)$-exclusion, and at most $7k\lceil \log_2(N/k)\rceil$ are required for the $(N-1, k)$-exclusion tree. Space complexity is dominated by the space required for the $(N-1, k)$-exclusion tree, which by Theorem 8 is $\Theta(N)$. Thus, we have the following result.

Theorem 9: Using fetch-and-add and test-and-set, (N, k)-exclusion can be implemented on a cache-coherent machine with remote-reference count 9 in the absence of contention, and $7k\lceil \log_2(N/k)\rceil + 8$ under contention, and with space complexity $\Theta(N)$. $\square$

A second alternative is to implement $(N-1, k)$-exclusion inductively using the algorithm given in Figure 6.6, as depicted inside the dotted box in Figure 6.7. This approach is designed to achieve performance that degrades gracefully with increasing contention, rather than performance that drops suddenly when contention rises. In particular, if contention is at most c, then a process accesses at most c instances of $(k+1,k)$-exclusion, each of which generates at most 7 remote memory references. A process that accesses c instances of $(k+1,k)$-exclusion also performs $c-1$ unsuccessful test-and-set operations and one successful test-and-set (if $c < N$), and clears the bit it successfully sets. This gives a total of $8c+1$ remote memory references. Note that this approach uses $N-k$ instances of $(k+1,k)$-exclusion, and $N-k+1$ test-and-set bits. By Theorem 7, each instance of $(k+1,k)$-exclusion requires $\Theta(1)$ space. Thus, we have the following.

Theorem 10: Using fetch-and-add and test-and-set, (N,k)-exclusion can be implemented on a cache-coherent machine with remote-reference count $8c+1$ if contention is at most c, and with space complexity $\Theta(N)$. □

The following corollaries regarding k-assignment follow from Theorems 9 and 10 and Lemma 1.

Corollary 1: Using fetch-and-add and test-and-set, (N,k)-assignment can be implemented on a cache coherent machine with remote-reference count 11 in the absence of contention, and $7k\lceil\log_2(N/k)\rceil + k + 8$ under contention, and with space complexity $\Theta(N)$. □

Corollary 2: Using fetch-and-add and test-and-set, (N,k)-assignment can be implemented on a cache-coherent machine with remote-reference count $9c+2$ if contention is at most c, and with space complexity $\Theta(N)$. □

6.4 k-Exclusion on Distributed Shared-Memory Machines

In the previous section, we showed that k-exclusion can be efficiently implemented on cache-coherent machines. Such implementations are efficient because when a process waits on a variable, that variable migrates to a local cache line. A distributed shared-memory machine without cache-coherence does not provide this luxury. On such a machine, each variable is local to only one processor, so for good scalability, different processes must wait on different variables. This makes k-exclusion more difficult to implement efficiently. Nevertheless, in this section, we show that (N, k)-exclusion can be implemented efficiently on distributed shared-memory machines without coherent caches. This is achieved by designing algorithms in which all busy waiting is performed on locally-accessible shared variables that are statically allocated to processes. The algorithms presented here are much simpler and more efficient than the distributed shared-memory algorithms we presented in [8].

Our approach here is the same as that of Section 6.3. In particular, we inductively reduce the problem of implementing (N, k)-exclusion to that of implementing $(k+1, k)$-exclusion. We present two algorithms for $(k+1, k)$-exclusion, both of which have constant remote-reference counts. The two algorithms differ in space complexity. The first algorithm uses fetch-and-add and test-and-set. Although this algorithm is efficient, it can lead to high space complexity when used inductively. In particular, each process needs a distinct spin location for each instance of $(n+1, n)$-exclusion, where $k \leq n < N$, in an inductive application. The second algorithm we present improves on this by being structured so that in inductive applications, each process uses only a constant number of spin locations across all instances of $(n+1, n)$-exclusion. This reduction in spin locations comes at the expense of using a third primitive, namely compare-and-swap. Both algorithms are based on the same intuition as that of the algorithm for cache-coherent machines in Figure 6.3. Specifically, we seek to implement a queue of size one to hold any blocked processes. However, the need to rely only on statically allocated local spin locations complicates matters slightly. The resulting correctness proofs are not hard, but are slightly more tedious than that given in the previous section, so we defer their presentation to Appendix B.

6.4.1 First Algorithm

Our first (N, k)-exclusion algorithm for distributed shared-memory machines is given in Figure 6.8. As before, we assume that the *Acquire* and *Release* procedures correctly implement $(N, k+1)$-exclusion, and that they use variables distinct from those in the remainder of the algorithm. Instead of all processes waiting on one spin location Q, each process p now has its own local spin location, $P[p]$.

The main difference between this algorithm and the one in Figure 6.3 is that spin locations are now separate from the queue Q. In the cache-coherent algorithm of Figure 6.3, a process can enqueue itself onto the queue, dequeue the previously-queued process, and end any spinning by the previously-queued process in one step by simply assigning its own process identifier to Q. In the algorithm shown in Figure 6.8, it takes several steps to accomplish all of this. Thus, we could potentially have a situation in which two or more processes both concurrently attempt to enqueue themselves onto Q.

Let us examine this possibility in more detail. When $k+1$ processes have successfully executed the *Acquire* procedure, it is required that at least one of these processes wait, so that k-Exclusion is not violated. Thus, when a process q releases a process p from

```
shared variable
    X: −1..k;
    Q: 0..N − 1;
    P: array[0..N − 1] of boolean                          /* P[p] is local to process p */
initially
    X = k ∧ Q = 0 ∧ ¬P[0] ∧ (∀i : 1 ≤ i < N :: P[i])

process p                                                            /* 0 ≤ p < N */
private variable
    v: 0..N − 1

while true do
0:  Noncritical Section;
1:  Acquire(N,k + 1);                          /* Entry section of (N,k + 1)-exclusion */
2:  if fetch_and_add(X, −1) = 0 then                            /* No slots available */
3:      v := Q;                                              /* Get current spin location */
4:      if ¬test_and_set(P[v]) then                  /* Release currently spinning process */
5:          Q := p;                                          /* Become waiting process */
6:          P[p] := false;                                  /* Initialize spin location */
7:          if X < 0 then                         /* Still no slots available - must wait */
8:              while ¬P[p] do /* null */ od                     /* Wait until released */
    fi fi fi;
9:  Critical Section;
10: fetch_and_add(X, 1);                                              /* Release a slot */
11: v := Q;                                                  /* Get current spin location */
12: if ¬test_and_set(P[v]) then                      /* Release currently spinning process */
13:     Q := p;                                    /* Pretend to become waiting process */
14:     P[p] := false                                       /* Initialize spin location */
    fi;
15: Release(N,k + 1)                            /* Exit section of (N,k + 1)-exclusion */
od
```

Figure 6.8: (N, k)-exclusion for distributed shared-memory machines.

its spin loop, process q should itself start waiting. Since it takes several steps for q to accomplish this, it is possible that before q releases p, another process r detects that p is in the queue, releases p, and starts waiting.[4] If q does not detect this, then q might start waiting too. If the $k-1$ remaining processes are faulty, then q and r might wait forever, violating Starvation-Freedom. Thus, we need a mechanism to allow process q to detect that process p has already been released. The test-and-set instruction in statement 4 of Figure

[4]To see that it is possible for this many processes to concurrently execute statements 3 through 8, consider the following. From a state in which one process is at statement 3, and k processes are in their critical sections, let one of the latter leave its critical section, and then attempt to enter it again. If no other process takes a step, then that process must reach statement 3. Continuing in this manner, we could reach a state in which $k+1$ processes are executing statements 3 through 8.

6.8 serves this purpose. Specifically, this statement ensures that at most one of q and r will release p from its spinning. This is the essential difference between the algorithm of Figure 6.8 and that of Figure 6.3.

With $Acquire(N, k+1)$ and $Release(N, k+1)$ replaced by skip statements, the algorithm in Figure 6.8 solves $(k+1, k)$-exclusion on distributed shared-memory machines with remote-reference count 9 (recall that $P[p]$ is local to process p). As before, in inductive applications of this algorithm, we require that different instances of the algorithm employ distinct shared variables. Because of the spin locations, $P[0], \ldots, P[N-1]$, each instance requires $\Theta(N)$ space (as compared to $\Theta(1)$ for the algorithm of Figure 6.3). Thus, we have the following counterpart to Theorem 7.

Theorem 11: Using fetch-and-add and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with remote-reference count $9(N-k)$ and with space complexity $\Theta(N^2)$. □

The tree-based approach of Figure 6.4(a) can be applied here as well, yielding the following counterpart to Theorem 8.

Theorem 12: Using fetch-and-add and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with remote-reference count $9k\lceil \log_2(N/k) \rceil$ and with space complexity $\Theta(N^2)$. □

The two fast-path approaches described in Section 6.3 can be also be used. Hence, we have the following counterparts to Theorems 9 and 10.

Theorem 13: Using fetch-and-add and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with remote-reference count 11 in the absence of contention, and $9k\lceil \log_2(N/k)\rceil + 10$ under contention, and with space complexity $\Theta(N^2)$.□

Theorem 14: Using fetch-and-add and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with remote-reference count $10c + 1$ if contention is at most c, and with space complexity $\Theta(N^2)$. □

The remote-reference counts in the above two theorems are calculated in the same way as was done prior to Theorems 9 and 10, but using 9 as the remote-reference count of $(k + 1, k)$-exclusion instead of 7.

The following corollaries regarding k-assignment follow from Theorems 13 and 14 and Lemma 1.

Corollary 3: Using fetch-and-add and test-and-set, (N, k)-assignment can be implemented on a distributed shared-memory machine with remote-reference count 13 in the absence of contention, and $9k\lceil \log_2(N/k)\rceil + k + 10$ under contention, and with space complexity $\Theta(N^2)$. □

Corollary 4: Using fetch-and-add and test-and-set, (N, k)-assignment can be implemented on a distributed shared-memory with remote-reference count $11c+2$ if contention is at most c, with space complexity $\Theta(N^2)$. □

6.4.2 Second Algorithm

Our first $(k+1, k)$-exclusion algorithm for distributed shared-memory machines has the disadvantage that in inductive applications each process needs a separate spin location for each instance of the algorithm. This can result in space complexity that is somewhat high. Our second $(k+1, k)$-exclusion algorithm for distributed shared-memory machines remedies this problem by allowing each process to use the same two spin locations across all instances of the algorithm in inductive applications. This algorithm is shown in Figure 6.9. The two spin locations of each process p are denoted $R[p]$ and $P[p]$.

Before examining the code in Figure 6.9 in detail, we first consider the pitfalls involved in using a constant number of spin locations per process in inductive applications. Specifically, consider an inductive application of the algorithm in Figure 6.9, and suppose that each process uses the same R and P variables in all instances of this algorithm (as in our previous algorithms, we assume that other shared variables are not used in different instances). Then, to make sure that such an inductive application is correct, in our correctness proof for the algorithm of Figure 6.9, we need to allow for the possibility that $R[p]$ and $P[p]$ may be modified in the noncritical section, $Acquire(N, k+1)$ procedure, critical section, or $Release(N, k+1)$ procedure of p or some other process. This is because these sections and procedures may in fact contain other instances of the algorithm in Figure 6.9 that may modify these variables.

It should be clear from the preceding paragraph that, the correctness proof for the algorithm of Figure 6.9 must allow for the possibility that $R[p]$ and $P[p]$ may be modified by some statement in the noncritical section, $Acquire(N, k+1)$ procedure, critical section, or $Release(N, k+1)$ procedure of p or some other process. However, note that in inductive applications of this algorithm, these variables may be modified in these sections and procedures only by executing code like that in Figure 6.9. If we assign distinct *instance numbers* — the IN constant in Figure 6.9 — to different instances of this code, then the modifica-

```
constant
    IN: an integer value
        /* In inductive applications, different (k+1, k)-exclusion instances use distinct IN numbers */

type
    Spintype = record  flag: boolean;  instance: integer  end

shared variable
    X: -1..k;
    Z: boolean;
    Q: 0..N-1;
    R, P: array[0..N-1] of Spintype                    /* R[p] and P[p] are local to process p */
 /* In inductive applications, each (k+1, k)-exclusion instance uses distinct X, Z, and Q variables, ... */
                                  /* ... but R[p] and P[p] are shared across all (k+1, k)-exclusion instances */
initially
    X = k ∧ Z = false ∧ Q = 0

process p                                                                        /* 0 ≤ p < N */
private variable
    v: 0..N-1

while true do
    Noncritical Section;
    Acquire(N, k+1);                                          /* Entry section of (N, k+1)-exclusion */
    if fetch_and_add(X, -1) = 0 then                                      /* No slots available */
        if ¬test_and_set(Z) then
            v := Q;                                                   /* Get current spin location */
            compare_and_swap(P[v], (false, IN), (true, IN));   /* Release currently spinning process */
            Q := p;                                                      /* Become waiting process */
            P[p] := (false, IN);                                   /* Initialize "first" spin location */
            Z := false;
            R[p] := (false, IN);                                  /* Initialize "second" spin location */
            if X < 0 then                                  /* Still no slots available - must wait */
                while P[p] = (false, IN) ∧
                      R[p] = (false, IN) do /* null */ od                     /* Wait until released */
    fi fi fi;
    Critical Section;
    fetch_and_add(X, 1);                                                           /* Release a slot */
    v := Q;                                                           /* Get current spin location */
    compare_and_swap(R[v], (false, IN), (true, IN));           /* Release currently spinning process */
    Release(N, k+1)                                             /* Exit section of (N, k+1)-exclusion */
od
```

Figure 6.9: Space-efficient (N, k)-exclusion for distributed shared-memory machines.

tions to spin locations in one instance will not adversely interfere with another instance. In the following lemma, we use Hoare triples to enumerate the kinds of interferences that can occur.

Lemma 2: For any statement s in the noncritical or critical section, or $Acquire(N, k+1)$ or $Release(N, k+1)$ procedure of any process q, the following properties hold, where b ranges over $\{false, true\}$.

$$\{p \neq q \ \wedge \ P[p] = (b, IN)\} \ q.s \ \{P[p] = (b, IN)\} \tag{S1}$$

$$\{p \neq q \ \wedge \ R[p] = (b, IN)\} \ q.s \ \{R[p] = (b, IN)\} \tag{S2}$$

$$\{P[q] = (b, IN)\} \ q.s \ \{P[q] = (b, IN) \ \vee \ P[q].instance \neq IN\} \tag{S3}$$

$$\{R[q] = (b, IN)\} \ q.s \ \{R[q] = (b, IN) \ \vee \ R[q].instance \neq IN\} \tag{S4}$$

$$\{P[p].instance \neq IN\} \ q.s \ \{P[p].instance \neq IN\} \tag{S5}$$

$$\{R[p].instance \neq IN\} \ q.s \ \{R[p].instance \neq IN\} \tag{S6}$$

Proof: Let statement s be as defined in the lemma. To see that property (S1) holds, note that $q.s$ can change $P[p]$ only by means of a compare-and-swap instruction in an instance other than IN. Hence, as different instances of the algorithm in Figure 6.9 have different instance numbers, s cannot falsify $P[p] = (b, IN)$ in this case, because if it attempted to do so, then its compare-and-swap would fail. Property (S2) holds for similar reasons. To see that property (S3) holds, note that, because different instances of the algorithm use different instance numbers, if $p.s$ modifies $P[p]$, then it establishes $P[p].instance \neq IN$. Property (S4) holds from similar reasons. Finally, to see that property (S5) holds, note that process p only establishes $P[p].instance = IN$ by executing statements within the instance of the algorithm with instance number IN. Also, no process $q \neq p$ can change the *instance* field of

$P[p]$. Property (S6) holds for similar reasons. $\square$

The discussion above explains much of the insight that underlies the algorithm of Figure 6.9. The remaining details are as follows. As in the algorithm of the previous subsection, we need to ensure that we do not end up enqueueing two processes at the same time. This is prevented by the test-and-set of Z at statement 3. This test-and-set ensures that at most one process at a time executes the code in statements 4 through 8. Note, however, that a process could enter this region of code and then fail before executing its compare-and-swap at statement 5, i.e., before freeing the currently-spinning process. To get around this potential problem, we use a second spin location that processes update in their exit sections. Thus, if a nonfailed process p is spinning at statements 11 and 12, and if another process executes a successful test-and-set at statement 3 but fails before executing its compare-and-swap at statement 5, then we can show that there is a nonfailed process that will eventually free p by executing its compare-and-swap at statement 16. This completes our informal description of this algorithm.

With $Acquire(N, k+1)$ and $Release(N, k+1)$ replaced by skip statements, the algorithm in Figure 6.9 solves $(k+1, k)$-exclusion on distributed shared-memory machines with remote-reference count 10 (recall the $P[p]$ and $R[p]$ are both local to process p). As discussed above, in inductive applications of this algorithm, each process uses the same two spin locations across all instances of $(n+1, n)$-exclusion, where $k \leq n < N$. This gives a total of $\Theta(N)$ space for spin locations. In addition, constant additional space is required for each instance. Thus, if an inductive application uses M instances of $(n+1, n)$-exclusion, then its space complexity is $\Theta(N + M)$. With these observations in mind, we have the following counterparts to Theorems 11 through 14, and Corollaries 3 and 4 respectively.

Theorem 15: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with remote-reference count $10(N-k)$ and with space complexity $\Theta(N)$. $\square$

Theorem 16: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with remote-reference count $10k\lceil\log_2(N/k)\rceil$ and with space complexity $\Theta(N)$. $\square$

Theorem 17: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with remote-reference count 12 in the absence of contention, and $10k\lceil\log_2(N/k)\rceil + 11$ under contention, and with space complexity $\Theta(N)$. $\square$

Theorem 18: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with remote-reference count $11c + 1$ if contention is at most c, and with space complexity $\Theta(N)$. $\square$

Corollary 5: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-assignment can be implemented on a distributed shared-memory machine with remote-reference count 14 in the absence of contention, and $10k\lceil\log_2(N/k)\rceil + k + 11$ under contention, and with space complexity $\Theta(N)$. $\square$

Corollary 6: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-assignment can be implemented on a distributed shared-memory with remote-reference count $12c + 2$ if contention is at most c, with space complexity $\Theta(N)$. $\square$

6.5 Performance Results

In this section, we present results from experiments that we conducted to evaluate the performance of some of the algorithms presented in the previous sections. These experiments involve implementations of k-resilient shared objects in systems that are multiprogrammed.

Wait-free algorithms are designed to tolerate the delay of any process when *all* processes simultaneously access an object. However, as mentioned earlier, wait-freedom links resiliency to worst-case contention, and this can be overkill in practice. From a performance standpoint, linking resiliency to expected levels of contention may be preferable. However, doing so requires an approach for efficiently implementing shared objects that tolerate fewer than $N - 1$ failures, while incurring less overhead. As discussed earlier, this can be accomplished by combining a wait-free algorithm for k processes with a k-assignment algorithm.

In the following two subsections, we present results from performance experiments that compare k-resilient objects with wait-free and spin-lock-based object implementations. These k-resilient objects are implemented using one of our k-assignment algorithms, together with Herlihy's small-object, wait-free construction for the k-process wait-free implementation. The first subsection below contains results of experiments conducted on a cache-coherent multiprocessor, while the second contains results from experiments conducted on a distributed shared-memory multiprocessor without a cache-coherence mechanism. These experiments show that, for both classes of machines, k can be chosen so that k-resilient objects implemented using our algorithms are faster and scale better than objects implemented using either wait-free or lock-based algorithms. These results validate our earlier claims that our k-exclusion algorithms are fast and scalable.

All of our experiments have the same structure, so before we present any specific details, we give a brief overview. On both machines, we implemented a shared priority queue using the local-spin queue lock of Mellor-Crummey and Scott [72], and Herlihy's universal wait-free object construction [44]. To test the performance of our algorithms we used the "inductive fast path" method of Figure 6.6 with each level implemented using the appropriate algorithm (Figure 6.3 on the cache-coherent multiprocessor, and Figure 6.8 on the distributed shared-memory multiprocessor). In each experiment, a fixed number of priority queue operations are performed (50,000 on the cache-coherent multiprocessor, and 20,000 on the distributed shared-memory multiprocessor). The number of participating processes is varied, and the priority queue operations are equally divided among these processes. Previous experiments involving scalable synchronization constructs have assumed that each process runs on a dedicated processor [12, 38, 72, 97]. However, in practice it can be desirable to run more than one process on each processor. In our experiments, we consider scenarios in which processes share processors by multiprogramming. In order to test the performance of each method under varying levels of multiprogramming, we fix the number of processors and vary the number of processes. In each performance graph presented, we plot the total time taken to complete the operations using the various object implementations being compared. In the case of our approach, we show k-resilient implementations for varying values of k. An object implementation scales well if its total time does not increase much as the number of participating processes increases, i.e., if the curve plotted for that implementation is relatively flat.

6.5.1 Cache-Coherent Multiprocessors

The results presented in this section are taken from experiments run on a Sequent Symmetry multiprocessor. The Sequent Symmetry is a shared-memory multiprocessor whose processor and memory nodes are interconnected via a shared bus. A processor node consists of an Intel 80386 and a 64 Kbyte, two-way set-associative cache. Cache coherence is maintained by a snoopy protocol. The Symmetry provides an atomic fetch-and-store instruction. For these experiments, we simulated a simple, round-robin multiprogramming environment through the use of a dedicated processor to act as a scheduler. (Scheduling on the Sequent Symmetry is priority-based and is therefore not particularly representative of general multiprogramming environments.) Processes are preempted and rescheduled using Unix signals, and a preempted process waits using the Unix `sigpause()` system call. We simulated synchronization primitives that are not directly provided by using short critical sections. In order to closely simulate these primitives, they and the scheduler were designed so that a process would not be preempted while executing in one of these critical sections.

Figure 6.10 compares the performance of various implementations of a shared priority queue. First, observe that Mellor-Crummey and Scott's algorithm suffers severely under multiprogramming. This is because processes in this algorithm wait in a queue for access to the critical section. If a process in the queue is delayed due to preemption, then all the processes behind it in the queue are also delayed. Furthermore, while these processes are waiting for the delayed process, they are likely to be preempted themselves, exacerbating the problem even further.

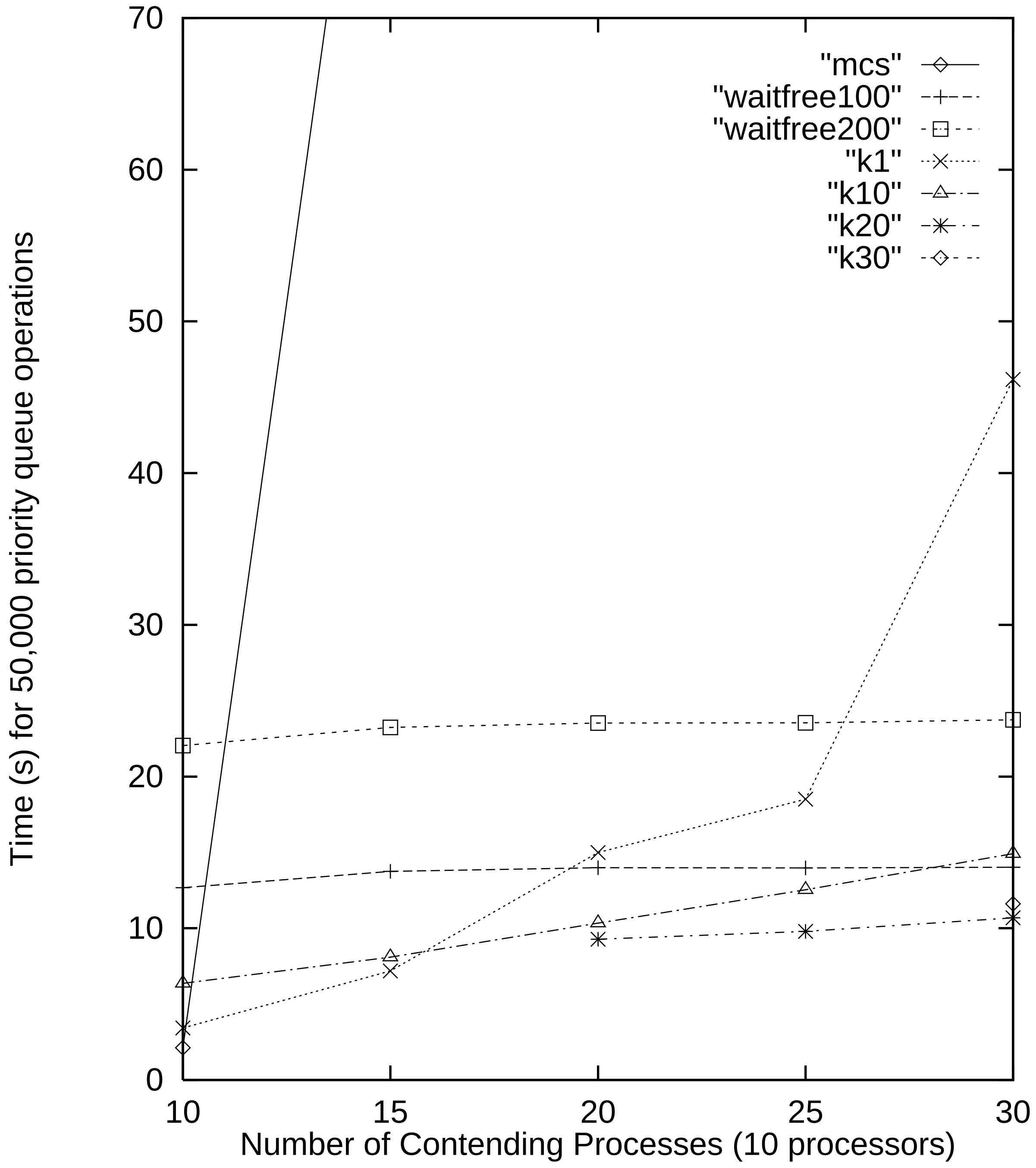


Figure 6.10: Performance Experiments on the Sequent Symmetry. Mellor-Crummey and Scott's algorithm is labeled "mcs"; Herlihy's algorithm for $N = 100$ and $N = 200$ are labeled "waitfree100" and "waitfree200", respectively; and our algorithms for $k = 1$, $k = 10$, $k = 20$, and $k = 30$ are labeled "k1", "k10", "k20", and "k30", respectively.

Figure 6.10 also shows the performance of Herlihy's wait-free algorithm in this setting. Two curves are shown for Herlihy's algorithm — one for a 100-process implementation and one for a 200-process implementation. (Note that these are not the actual number of processes participating in the experiments, but the maximum number of processes the implementation can accommodate.) It is interesting to note that the resiliency provided by Herlihy's algorithm allows it to outperform Mellor-Crummey and Scott's algorithm by a significant margin in both cases. However, as mentioned above, many wait-free shared object implementations, including Herlihy's, do not scale well as N — the total number of processes for which the object is implemented — increases. This is because they have time complexity that is at least proportional to N. This is demonstrated in the case of Herlihy's algorithm by the fact that the 100-process implementation outperforms the 200-process implementation, despite the fact that the same number of processes perform the same number of operations in each case. As discussed below, objects implemented using our algorithms do not suffer from this problem.

We now turn to the performance of objects implemented using the k-exclusion algorithm presented in Figure 6.3. Observe that, under multiprogramming, our algorithm performs significantly better than Mellor-Crummey and Scott's for all values of k shown, and also better than Herlihy's algorithm in most cases. Also observe that, as the level of multiprogramming increases, the performance of our approach is better for larger choices of k, that is, when a higher level of resilience is provided. However, when the highest level of resilience, i.e., wait-freedom, is used, performance is worse. This demonstrates the utility of algorithms such as ours that allow the level of resilience of a shared object implementation to be set according to system parameters.

It is interesting to note that when $k = 1$ — that is, when our algorithm is reduced to a mutual exclusion algorithm — it significantly outperforms Mellor-Crummey and Scott's mutual exclusion algorithm, and in some cases outperforms implementations (both Herlihy's and ours) that provide higher levels of resiliency. Two factors contribute to this good performance. First, our k-exclusion algorithm does not enforce a strict FIFO order on waiting processes. Thus, the bad behavior described above for Mellor-Crummey and Scott's algorithm is not exhibited by ours. Second, our k-exclusion algorithm does not need to provide any resiliency when $k = 1$, which makes it simpler than Herlihy's algorithm and simpler than our algorithm for higher values of k. Despite these advantages, however, our 1-exclusion algorithm can be seen to suffer under higher levels of multiprogramming. This is because, as more and more waiting processes are preempted, the likelihood of "chains" of waiting processes increases. Our algorithm outperforms Mellor-Crummey and Scott's because their algorithm is queue-based, which means these chains are much more likely to form, and once they have formed, are almost certain to remain. In contrast, our algorithm avoids these chains forming, and also allows the chains to disband when the preempted processes resume execution. This is achieved by allowing processes to enter the critical section despite the preemption of other waiting processes.

6.5.2 Distributed Shared-Memory Multiprocessors

The results presented in this section are taken from experiments run on a BBN GP1000 multiprocessor. The BBN GP1000 is a shared-memory multiprocessor whose processor and memory nodes are interconnected using a BBN Butterfly Switch. Each processor node consists of an Motorola 68020 with no cache. The GP1000 provides an atomic "clear-then-add" instruction, which can be used to directly implement test-and-set and fetch-and-add instructions. We simulated synchronization primitives that are not directly provided by using short critical sections. Unlike our experiments on the Sequent Symmetry, processes on the BBN GP1000 are multiprogrammed under kernel control. As a result, it is possible for a process to be preempted within the critical section of a simulated synchronization primitive. However, simulated primitives are used to a similar extent by all algorithms tested, so this problem does not unfairly penalize any of the algorithms.

We performed experiments on the BBN GP1000 similar to those described in the previous subsection. In this case, we used 40 processors, and varied the number of processes between 40 and 160. As mentioned above, we used the k-exclusion algorithm presented in Figure 6.8 to test the performance of our approach to shared object implementation. The results are presented in Figure 6.11.

Figure 6.11 shows trends for the relative performance of the three implementations that are similar to those shown for the Sequent Symmetry. Again, Mellor-Crummey and Scott's algorithm suffers severely under multiprogramming and Herlihy's algorithm performs much better. As before, two curves are shown for Herlihy's algorithm — this time one curve is for a 200-process implementation and one for a 400-process implementation. As in the Sequent Symmetry experiments, the performance of Herlihy's algorithm does not scale well as N increases. When $k = 1$, our algorithm is reduced to a mutual exclusion algorithm and outperforms all other approaches for most data points shown. However, as the multiprogramming level increases, the performance of this approach degrades, and

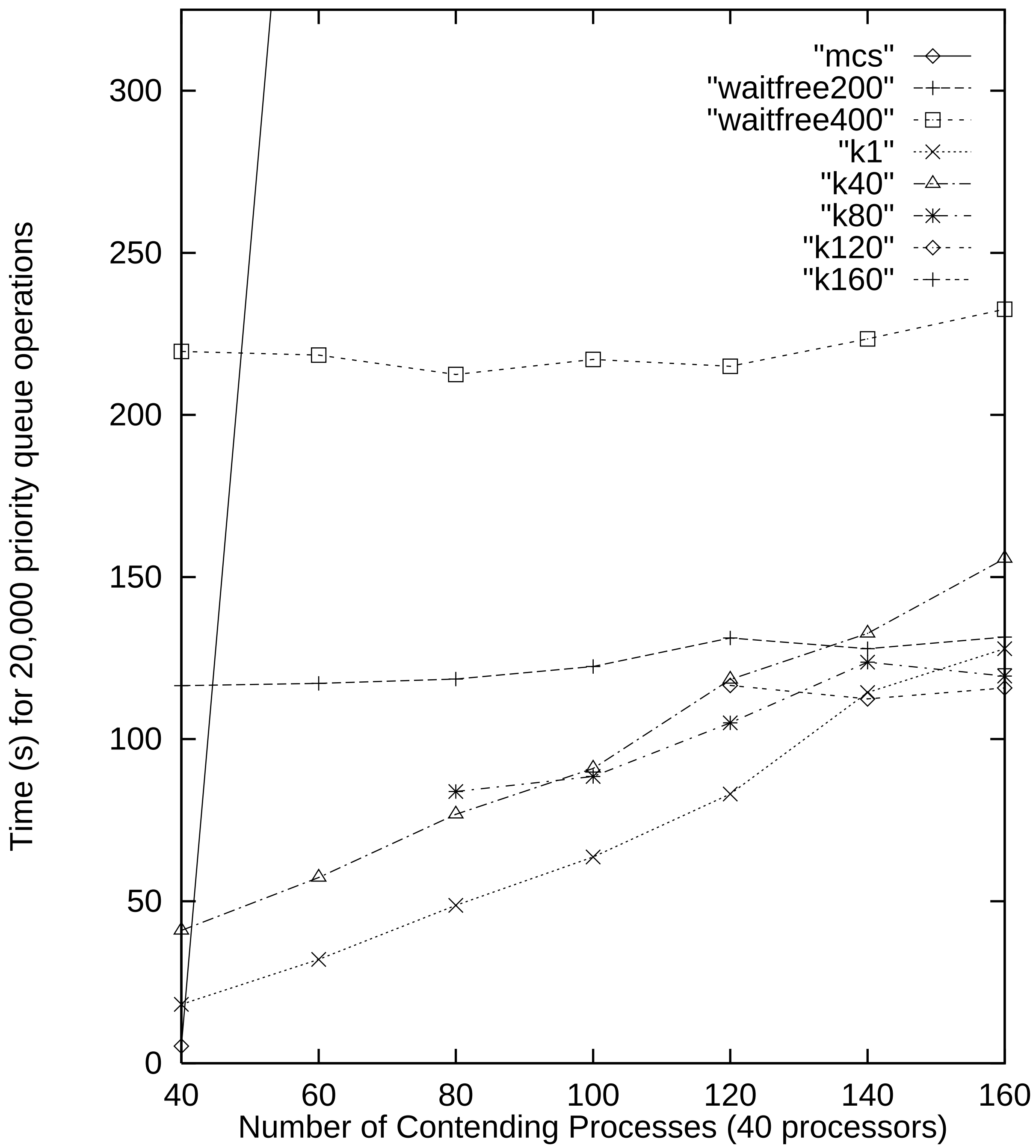


Figure 6.11: Performance Experiments on the BBN GP1000. Mellor-Crummey and Scott's algorithm is labeled "mcs"; Herlihy's algorithm for $N = 200$ and $N = 400$ are labeled "waitfree200" and "waitfree400", respectively; and our algorithms for $k = 1$, $k = 40$, $k = 80$, $k = 120$, and $k = 160$ are labeled "k1", "k40", "k80", "k120", and "k160", respectively.

by choosing k appropriately, the k-exclusion-based approach can be configured to perform better. The reason that the $k = 1$ case performs better on the BBN GP1000 than it did on the Sequent Symmetry relative to the other approaches is the increased relative cost of copying on the BBN GP1000. In order to tolerate delays, Herlihy's algorithm performs each operation on a copy of the object and later attempts to make this copy current. The need to make copies of the object adversely affects the performance of Herlihy's algorithm and therefore of our algorithms when $k > 1$. However, the increased cost of copying does not degrade performance when $k = 1$ because no copying is necessary in this case. This concludes the description of our performance experiments.

Chapter 7

Fast, Long-Lived Renaming

In the previous chapter, we used a simple, test-and-set-based, long-lived renaming algorithm to assign distinct names to processes inside the k-exclusion critical section for use in the k-process object implementation. In this chapter, we study the renaming problem in greater depth. In particular, we investigate the impact of the available instruction set on the problem of renaming.

Previous research on renaming has focused on the *one-time* renaming problem [14, 19, 23, 74], in which each of k processes is required to choose a distinct value, called a *name*, that ranges over $\{0, ..., M-1\}$. Each process is assumed to have a unique process identifier ranging over $\{0, ..., N-1\}$. It is further required that $k \leq M < N$. Thus, an M-renaming algorithm is invoked by k processes in order to reduce the size of their name space from N to M. One-time renaming is useful when processes perform a computation whose time complexity is dependent on the size of the name space containing the processes. By first using an efficient renaming algorithm to reduce the size of the name space, the time complexity of that computation can be made independent of the original name space.

In this chapter, we also consider *long-lived* renaming, a more general version of renaming in which processes may repeatedly acquire and release names. A solution to the long-lived renaming problem is useful if a set of processes repeatedly performs a computation whose time complexity is dependent on the size of the name space containing the processes that participate concurrently. As described in the previous chapter, the specific application that motivated us to study this problem is the implementation of resilient, scalable shared objects. The approach described there only restricts the number of processes that access the shared object implementation *concurrently*. Over time, many processes may access it. Thus, it is not sufficient for a process to simply acquire a name once and retain that name for future use: a process must be able to release its name so that another process may later acquire the same name.

The renaming problem has been studied previously for both message-passing [14] and shared-memory multiprocessing systems [19, 23, 26]. We present several wait-free algorithms for both one-time and long-lived renaming on shared-memory multiprocessing systems. Previous wait-free renaming algorithms have time complexity that is dependent on the size of the original name space. Thus, these algorithms suffer from the same shortcoming that the renaming problem is intended to overcome. In contrast, most of our algorithms have time complexity that is independent of the size of the original name space. In the remainder of this chapter, we call a renaming algorithm *fast* if the worst case time complexity of acquiring (and releasing, if long-lived) a name once is independent of N, and polynomial in k.

In this chapter, we present several new algorithms for one-time and long-lived renaming, one of which is a generalization of the algorithm presented in the previous chapter. We first present renaming algorithms that use only atomic read and write instructions. It has been shown that if $M < 2k-1$, then M-renaming cannot be implemented in a wait-free manner using only atomic reads and writes [46]. Wait-free, read/write algorithms for one-time renaming that yield an optimal name space of size $M = 2k-1$ have been proposed in [19, 23]. However, in these algorithms, the time complexity of choosing a name is dependent on N, the size of the original name space. Similarly, the only previous algorithm that solves long-lived renaming to a name space of size $2k-1$, due to Burns and Peterson[26],[1] has time complexity that is dependent on the size of the original name space.

We present the most efficient algorithm to date for fast, long-lived $k(k+1)/2$-renaming. To facilitate the presentation of this algorithm, we first present two read/write renaming algorithms, both of which yield a name space of size $k(k+1)/2$. The first is fast, but not long-lived, and the second is long-lived, but not fast. All of these algorithms employ a novel technique that uses "building blocks" based on the "fast path" mechanism employed by Lamport's fast mutual exclusion algorithm [65].

After presenting our fast, long-lived $k(k+1)/2$-renaming algorithm, we show how it can be combined with previous, non-fast algorithms to achieve fast, long-lived renaming to the optimal name space size of $2k-1$. New and previous read/write renaming algorithms are summarized in chronological order in Table 7.1.

[1] Actually, Burns and Peterson solved a more general problem, which they called ℓ-assignment. An ℓ-assignment protocol not only assigns names to processes, but also forces some processes to wait if too many request names concurrently. Nonetheless, if at most k processes access an ℓ-assignment protocol that guarantees that, provided at most $k-1$ processes are faulty, every process eventually gets a name, then none have to wait, so ℓ-assignment provides a wait-free solution to the long-lived ℓ-renaming problem. We are grateful to Hagit Attiya for pointing this out to us.

Reference	M	Time Complexity	Space Complexity	Fast?	Long-Lived?
[26]	$2k-1$	$\Theta(Nk^2)$	$\Theta(N^2)$	No	Yes
[19]	$k(k+1)/2$	$\Theta(Nk)$	$\Theta(N)$	No	No
[19]	$2k-1$	$\Theta(N4^k)$	$\Theta(N)$	No	No
[23]	$2k-1$	$\Theta(Nk^2)$	$\Theta(N^2)$	No	No
Thm. 19	$k(k+1)/2$	$\Theta(k)$	$\Theta(k^2)$	Yes	No
Thm. 20	$k(k+1)/2$	$\Theta(Nk)$	$\Theta(Nk^2)$	No	Yes
[24]	$k(k+1)/2$	$\Theta(k^3)$	$\Theta(k^4 \min(3^k, N))$	Yes	Yes
Thm. 21	$k(k+1)/2$	$\Theta(k^2)$	$\Theta(k^3)$	Yes	Yes
Thm. 22	$2k-1$	$\Theta(k^4)$	$\Theta(k^4)$	Yes	Yes

Table 7.1: A summary of read/write, wait-free M-renaming algorithms. Time complexity is the worst-case time complexity of acquiring (and releasing, if long-lived) a name once.

In the second part of this chapter, we consider long-lived k-renaming algorithms. By definition, M-renaming for $M < k$ is impossible, so with respect to the size of the name space, k-renaming is optimal. As mentioned previously, it is impossible to implement k-renaming using only atomic read and write operations. Thus, all of our k-renaming algorithms employ stronger, read-modify-write operations.

We present three wait-free, long-lived k-renaming algorithms. The first such algorithm uses two read-modify-write operations, *set_first_zero* and *clr_bit*. The *set_first_zero* operation is applied to a b-bit shared variable X whose bits are indexed from 0 to $b-1$. If some bit of X is clear, then *set_first_zero*(X) sets the first clear bit of X, and returns its index. If all bits of X are set, then *set_first_zero*(X) leaves X unchanged and returns b. Note that for $b = 1$, *set_first_zero* is equivalent to *test_and_set*. The *set_first_zero* operation for $b > 1$ can be implemented, for example, using the *atomff0andset* operation available on the BBN TC2000 multiprocessor [20]. The *clr_bit*(X, i) operation clears the ith bit of the b-bit shared variable X. For $b = 1$, *clr_bit* is a simple write operation. For $b > 1$, *clr_bit* can be implemented, using the *fetch_and_and* operation available on the BBN TC2000.

Reference	Time Complexity	Bits per Variable	Instructions Used
Thm. 23	$\Theta(k)$	1	write and test_and_set
Thm. 23	$\Theta(k/b)$	b	set_first_zero and clr_bit
Thm. 24	$\Theta(\log k)$	$\Theta(\log k)$	bounded_decrement and fetch_and_add
Thm. 25	$\Theta(\log(k/b))$	$\Theta(\log k)$	above, set_first_zero, and clr_bit

Table 7.2: A summary of wait-free, long-lived k-renaming algorithms. Time complexity is the worst-case time complexity of acquiring and releasing a name once.

Our second long-lived k-renaming algorithm employs the commonly-available *fetch_and_add* operation and the *bounded_decrement* operation. The *bounded_decrement* operation is similar to *fetch_and_add*$(X, -1)$, except that *bounded_decrement* does not modify a variable whose value is zero. We do not know of any systems that provide *bounded_decrement* as a primitive operation. However, at the end of Section 7.4, we show that *bounded_decrement* can be approximated in a lock-free manner using the *fetch_and_add* operation. This allows us to obtain a lock-free, long-lived k-renaming algorithm based on *fetch_and_add*. A renaming algorithm is *lock-free* if and only if it is guaranteed that each attempt by some process p to acquire or release a name terminates unless some other process acquires and releases a name infinitely often.

Our third long-lived k-renaming algorithm combines both algorithms discussed above, improving on the performance of each. Our wait-free, long-lived k-renaming algorithms are summarized in Table 7.2.

The remainder of this chapter is organized as follows. Section 7.1 contains definitions used in the rest of the chapter. In Sections 7.2 and 7.3, we present one-time and long-lived renaming algorithms that employ only atomic reads and writes. In Section 7.4, we present long-lived renaming algorithms that employ stronger read-modify-write opera-

```
process p                                                              /* 0 ≤ p < N */
private variable name : 0..M − 1                                       /* Name received */
  while true do
    Remainder Section;              /* Ensure at most k processes rename concurrently */
    Getname Section;                /* Assigns a value ranging over {0, ..., M − 1} to p.name */
    Working Section;
    Putname Section                                           /* Release the name obtained */
  od
```

Figure 7.1: Organization of processes accessing a long-lived renaming algorithm.

tions. Correctness proofs for algorithms in this chapter appear in Appendix C. (Because the results of Section 7.3.2 subsume the results of Sections 7.2 and 7.3.1, and because full correctness proofs for those results are presented elsewhere [75], these proofs are not repeated here.)

7.1 Definitions

In the *one-time M-renaming* problem, each of k processes, with distinct process identifiers ranging over $\{0, ..., N-1\}$, chooses a distinct value ranging over $\{0, ..., M-1\}$. We assume that $1 < k \leq M < N$. A solution to the M-renaming problem consists of a wait-free code fragment for each process p that assigns a value ranging over $\{0, ..., M-1\}$ to a private variable $p.name$ and then halts. For $p \neq q$, the same value should not be assigned to both $p.name$ and $q.name$.

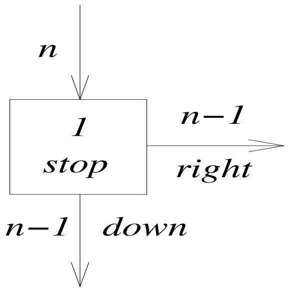
In the *long-lived M-renaming* problem, each of N distinct processes repeatedly executes a *remainder section*, acquires a name by executing a *getname section*, uses that name in a *working section*, and then releases the name by executing a *putname section*. The organization of these processes is shown in Figure 7.1. It is assumed that each process is initially in its remainder section, and that the remainder section guarantees that at most

k processes are outside their remainder sections at any time. A solution to the long-lived M-renaming problem consists of wait-free code fragments that implement the getname and putname sections shown in Figure 7.1, along with associated shared variables. The getname section for process p is required to assign a value ranging over $\{0, ..., M-1\}$ to $p.name$. If distinct processes p and q are in their working sections, then it is required that $p.name \neq q.name$.

As discussed earlier, our algorithms use the *set_first_zero*, *clr_bit*, and *bounded_decrement* operations, among other well-known operations. We define these operations formally by the following atomic code fragments, where X is a b-bit shared variable whose bits are indexed from 0 to $b-1$, and Y is a non-negative integer. We stress that these code fragments are definitions, and should not be interpreted as implementations of the given operations.

$$
\begin{array}{lll}
\textit{set\_first\_zero}(X) & \equiv & \textbf{if } (\exists n : 0 \leq n < b :: \neg X[n]) \textbf{ then} \\
& & \quad m := (\min n : 0 \leq n < b :: \neg X[n]);\ \ X[m] := \textit{true};\ \ \textbf{return } m \\
& & \textbf{else} \\
& & \quad \textbf{return } b \\
& & \textbf{fi} \\
\textit{clr\_bit}(X, i) & \equiv & X[i] := \textit{false} \\
\textit{bounded\_decrement}(Y) & \equiv & m := Y;\ \ \textbf{if } Y \neq 0 \textbf{ then } Y := Y - 1 \textbf{ fi};\ \ \textbf{return } m
\end{array}
$$

In each of our algorithms, each atomically-accessible shared variable can be stored in one machine word for all reasonable values of N. For example, our read/write algorithms require shared variables of approximately $\log_2 N$ bits. Thus, on a 32-bit shared-memory multiprocessor, these shared variables can be accessed with one shared variable access if $N < 2^{32}$. We measure the time complexity of our algorithms in terms of the worst-case time complexity of acquiring (and releasing, if long-lived) a name once.



shared variable $X : \{\perp\} \cup \{0..N-1\};$
$\quad Y$: **boolean**
initially $X = \perp \wedge Y = \mathit{false}$

private variable $\mathit{move} : \{\mathit{stop}, \mathit{right}, \mathit{down}\}$

$X := p;$
if Y **then** $\mathit{move} := \mathit{right}$
else
$\quad Y := \mathit{true};$
$\quad$ **if** $X = p$ **then** $\mathit{move} := \mathit{stop}$
$\quad\quad$ **else** $\mathit{move} := \mathit{down}$
$\quad$ **fi**
fi

Figure 7.2: The one-time building block and the code fragment that implements it.

7.2 One-Time Renaming using Reads and Writes

In this section, we present a one-time $(k(k+1)/2)$-renaming algorithm that employs only atomic read and write operations. This algorithm serves to introduce the main ideas of our first long-lived renaming algorithm. It is also of interest in its own right, because it has $\Theta(k)$ time complexity: a significant improvement over previous read/write algorithms for one-time renaming. Our one-time $(k(k+1)/2)$-renaming algorithm is based on a "building block", which we describe next.

7.2.1 The One-Time Building Block

The one-time building block, depicted in Figure 7.2, is in the form of a wait-free code fragment that assigns to a private variable *move* one of three values: *stop*, *right*, or *down*. If each of n processes executes this code fragment at most once, then at most one process receives a value of *stop*, at most $n-1$ processes receive a value of *right*, and at most $n-1$ processes receive a value of *down*. We say that a process that receives a value of *down* "goes down", a process that receives a value of *right* "goes right", and a process that receives a value of *stop* "stops". Figure 7.2 shows n processes accessing a building block, and the maximum number of processes that receive each value.

The code fragment shown in Figure 7.2 shows how the building block can be implemented using atomic read and write operations. The technique employed is essentially that of the "fast path" mechanism used in Lamport's fast mutual exclusion algorithm [65]. A process that stops corresponds to a process successfully "taking the fast path" in Lamport's algorithm. The value assigned to *move* by a process p that fails to "take the fast path" is determined by the branch p takes: if p detects that Y holds, then p goes right, and if p detects that $X \neq p$ holds, then p goes down.

To see why the code fragment shown in Figure 7.2 satisfies the requirements of our building block, first note that it is impossible for all n processes to go right — a process can go right only if another process previously assigned $Y := true$. Second, the last process p to assign $X := p$ cannot go down because if it tests X, then it detects that $X = p$ and therefore stops. Thus, it is impossible for all n processes to go down. Finally, because Lamport's algorithm prevents more than one processes from "taking the fast path", it is impossible for more than one process to stop.

We show below how these building blocks can be used to solve the renaming problem. The basic approach is to use such building blocks to "split" processes into successively smaller groups. Because at most one process stops at any particular building block, a process that stops can be given a unique name associated with that building block. Furthermore, when the size of a group has been decreased enough times that at most one process remains, that process (if it exists) can be given a name immediately.

7.2.2 Using the One-Time Building Block to Solve Renaming

In this section, we use $k(k-1)/2$ one-time building blocks arranged in a "grid" to solve one-time $k(k+1)/2$-renaming for k processes. This approach is depicted in Figure 7.3 for $k = 5$. In order to acquire a name, a process p accesses the building block at the top left corner of the grid. If p receives a value of *stop*, then p acquires the name associated with that building block. Otherwise, p moves either right or down in the grid, according to the value received. This is repeated until p receives a value of *stop* at some building block, or p has accessed $k-1$ building blocks. The name returned is calculated based on p's final position in the grid. In Figure 7.3, each grid position is labeled with the name associated with that position. Because no process takes more than $k-1$ steps, only the upper left triangle of the grid is used, as shown in Figure 7.3.

The algorithm is presented more formally in Figure 7.4. Note that each building block in the grid is implemented using the code fragment shown in Figure 7.2. At most one process stops at each building block, so a process that stops at a building block receives a unique name. However, a process may also obtain a name by taking $k-1$ steps in the grid. In [75], we show that distinct processes that take $k-1$ steps in the grid acquire distinct names. We also prove that each process acquires a name from $\{0, ..., k(k+1)/2-1\}$ with worst-case time complexity $\Theta(k)$. Thus, we have the following result.

Theorem 19: Using *read* and *write*, wait-free, one-time $(k(k+1)/2)$-renaming can be implemented with worst-case time complexity $\Theta(k)$. $\square$

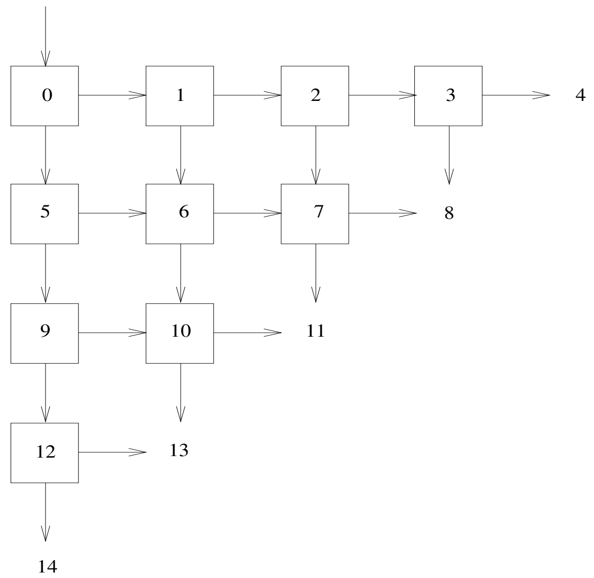


Figure 7.3: $k(k-1)/2$ building blocks in a grid, depicted for $k = 5$.

```
shared variable   X : array[0..k − 2, 0..k − 2] of {⊥} ∪ {0..N − 1};
                  Y : array[0..k − 2, 0..k − 2] of boolean
initially (∀r, c : 0 ≤ r < k − 1 ∧ 0 ≤ c < k − 1 :: X[r, c] = ⊥ ∧ Y[r, c] ≠ false)

process p                             /* k distinct processes ranging over 0..N − 1 */
private variable  name : 0..k(k + 1)/2 − 1;
                  stop : boolean;
                  i, j : 0..k − 1
initially i = 0 ∧ j = 0 ∧ ¬stop

    while i + j < k − 1 ∧ ¬stop do   /* Move down or across grid until stopping or reaching edge */
0:      X[i, j] := p;
1:      if Y[i, j] then j := j + 1                                     /* Move right */
        else
2:          Y[i, j] := true;
3:          if X[i, j] = p then stop := true else i := i + 1 fi    /* Stop or move down */
        fi
    od;
4:  name := ik − i(i − 1)/2 + j;              /* Calculate name based on position in grid */
5:  halt                                                /* Preserves p@5; has no effect */
```

Figure 7.4: One-time renaming using a grid of building blocks.

7.3 Long-Lived Renaming using Reads and Writes

In this section, we present two long-lived renaming algorithms that use only atomic read and write operations. These algorithms are both based on the grid algorithm presented in the previous section. In each case, we enable processes to release names as well as acquire names, by modifying the one-time building block. These modifications allow processes to "reset" a building block that they have previously accessed. This first algorithm yields a name space of size $k(k+1)/2$ and has time complexity $\Theta(Nk)$. (The time complexity of a long-lived renaming algorithm is the worst-case time complexity for acquiring and releasing a name once.) The second algorithm improves on this by reducing the time complexity to $\Theta(k^2)$, thereby achieving fast, long-lived $k(k+1)/2$-renaming. We now give an informal description of the first algorithm, which is proved correct in [75]. We provide a full correctness proof for the second algorithm in Appendix C.

7.3.1 Using a Long-Lived Building Block for Long-Lived Renaming

Our first long-lived renaming algorithm based on reads and writes is shown in Figure 7.5. As in the one-time algorithm presented in the previous section, a process acquires a name by starting at the top left corner of a grid of building blocks, and by moving through the grid according to the value received from each building block. The building blocks are similar to those described in the previous section, except that they can be "reset" (statement 6) after being accessed (statements 2 through 5). There are two significant differences between this algorithm and the one-time renaming algorithm.

Firstly, the single Y-bit in each building block of the one-time algorithm is replaced

```
shared variable   X : array[0..k−2, 0..k−2] of {⊥} ∪ {0..N−1};
                  Y : array[0..k−2, 0..k−2] of array[0..N−1] of boolean
initially (∀r, c, p : 0 ≤ r < k−1 ∧ 0 ≤ c < k−1 ∧ 0 ≤ p < N :: X[r,c] = ⊥ ∧ Y[r,c][p] = false)

process p                                                        /* 0 ≤ p < N */
private variable  name : 0..k(k+1)/2 − 1;
                  move : {stop, right, down};
                  i, j : 0..k−1;   h : 0..N
initially i = 0 ∧ j = 0 ∧ move = down

    while true do
0:     Remainder Section;
1:     i, j, move := 0, 0, down;                  /* Start at top left building block in grid */
       while i + j < k − 1 ∧ move ≠ stop do   /* Move through grid until stopping or reaching edge */
2:        X[i,j], h, move := p, 0, stop;         /* Will stop unless move later becomes right or down */
          while h < N ∧ move ≠ right do
3:           if Y[i,j][h] then move := right else h := h + 1 fi
          od;
4:        if move ≠ right then
             Y[i,j][p] := true;
5:           if X[i,j] ≠ p then move := down else move := stop fi
          fi;
6:        if move ≠ stop then
             Y[i,j][p] := false;                       /* Reset block if we didn't stop at it */
             if move = down then i := i + 1 else j := j + 1 fi     /* Move according to move */
          fi
       od;
7:     name := ik − i(i − 1)/2 + j;               /* Calculate name based on position in grid */
       Working Section;
8:     if i + j < k − 1 then                         /* If we stopped on a building block ... */
          Y[i,j][p] := false                          /* ... then reset that building block */
       fi
    od
```

Figure 7.5: Long-lived renaming with $\Theta(k^2)$ name space and $\Theta(Nk)$ time complexity.

by N Y-bits — one for each process. Instead of setting a common Y-bit, each process p sets a distinct bit $Y[p]$ (statement 4). This modification allows a process to reset the building block by clearing its Y-bit. A process resets a building block it has accessed before proceeding to the next building block in the grid (statement 6), or when releasing the name associated with that building block (statement 8). The building blocks are reset to allow processes to reuse the grid to acquire names repeatedly.

To see why N Y-bits are used, observe that in the one-time building block, the Y-variable is never reset, so using a single bit suffices. However, if only one Y-bit is used in the long-lived algorithm, a process might reset Y immediately after another process, say p, sets Y. Because the value p assigned to Y is overwritten, another process q may subsequently access the building block and fail to detect that p has accessed the building block. In this case, p and q may both receive a value of *stop* from the same building block, and consequently be assigned the same name.

The second difference between the one-time and long-lived building blocks is that they differ in time complexity. Instead of reading a single Y-variable for each block it encounters, each process now reads up to N Y-bits. This results in $\Theta(N)$ time complexity for accessing each long-lived building block. It may seem that all N Y-bits should be read in an atomic "snapshot" because, for example, p's write to $Y[p]$ might occur concurrently with q's scan of the Y-bits. In fact, this is unnecessary, because the fact that these operations are concurrent is sufficient to ensure that either p or q will not receive a value of *stop* from the building block.

In [75], we prove that, for distinct processes p and q, if $p@8 \ \wedge \ q@8$ holds, then

p and q hold distinct names from $\{0, ..., k(k+1)/2 - 1\}$. We also prove that the worst-case time complexity of acquiring and releasing a name once is $\Theta(Nk)$. Thus, we have the following result.

Theorem 20: Using *read* and *write*, wait-free, long-lived $(k(k+1)/2)$-renaming can be implemented so that the worst-case time complexity of acquiring and releasing a name once is $\Theta(Nk)$. $\square$

7.3.2 Making the Long-Lived Building Block Fast

We now present our second algorithm for fast, long-lived $(k(k+1)/2)$-renaming. Like the algorithms presented in Sections 7.2 and 7.3.1, this algorithm is based on a grid of building blocks, like the one shown in Figure 7.3. To facilitate formal proofs, we have incorporated all the building blocks into a single algorithm, shown in Figure 7.6. However, the building block structure should still be apparent. In lines 1 through 7, process p accesses building block (r, c), where $r = p.i$ and $c = p.j$. Building block (r, c) is made up of $X[r, c]$ and $Y[r, c, 0]$ through $Y[r, c, k - r - c]$.

As in the previous algorithms, a process acquires a name by starting at the top-left corner of the grid (line 0), and by moving through the grid according to the result of accessing each building block. At line 1, process p writes p into $X[p.i, p.j]$. At line 2, if process p reads $Y[p.i, p.j, n] \neq \perp$ for some $n < k - p.i - p.j$, then p moves right in the grid. If p always reads $X[p.i, p.j, p.h] = p$ at line 3 (so no process has overwritten the value written by p at line 1), then p writes p into $Y[p.i, p.j, n]$ for all $n \leq k - p.i - p.j$, and then reaches line 8 and acquires the name associated with building block $(p.i, p.j)$. On the other hand, if

shared variable X : **array**$[0..k-1, 0..k-1]$ **of** $\{\perp\} \cup \{0..N-1\}$;
Y : **array**$[0..k-1, 0..k-1, 0..k]$ **of** $\{\perp\} \cup \{0..N-1\}$
initially $(\forall r, c, n : 0 \leq r < k-1 \ \wedge\ 0 \leq c < k-1 \ \wedge\ r+c < k-1 \ \wedge\ 0 \leq n \leq k-r-c :: Y[r,c,n] = \perp)$

private variable *name*: $0..k(k+1)/2-1$; *moved*: **boolean**; i, j: $0..k-1$; h: $0..k$
initially $i = 0 \ \wedge\ j = 0 \ \wedge\ h = 0$

```
    while true do
0:    Remainder Section;
      i, j, moved := 0, 0, true;
      while i + j < k − 1 ∧ moved do
1:      X[i, j], h, moved := p, 0, false;
        while h < k − i − j ∧ ¬moved do
2:        if Y[i, j, h] ≠ ⊥ then
            j, moved := j + 1, true
          else h := h + 1
          fi
        od;
        h := 0;
        while h ≤ k − i − j ∧ ¬moved do
3:        if X[i, j] = p then
4:          Y[i, j, h], h := p, h + 1
          else
            while h > 0 do
5:            h := h − 1;
              if Y[i, j, h] = p then
6:              Y[i, j, h] := ⊥
              fi
            od;
7:          moved, i := true, i + 1
          fi
        od
      od;
8:    name := ik − i(i − 1)/2 + j;
9:    Working Section;
      if ¬moved then
        while h > 0 do
10:       h := h − 1;
          if Y[i, j, h] = p then
11:         Y[i, j, h] := ⊥
          fi
        od
      fi
    od
```

Figure 7.6: Long-lived renaming with $\Theta(k^2)$ name space, $\Theta(k^2)$ time complexity, and $\Theta(k^3)$ space complexity.

p reads $X[p.i, p.j] \neq p$ at line 3, then p executes lines 5 and 6 to reset the Y components it previously set, and then, at line 7, moves down in the grid. If process p takes $k-1$ "steps" in the grid, then it exits the loop beginning before line 1, and acquires a name. We later show that, in either case, no other process acquires the same name concurrently. Having described the overall structure of the algorithm, we now concentrate on how one building block is implemented.

The long-lived building block used in Section 7.3.1 uses N Y-bits — one for each process. This ensures that, if a process p sets the Y variable (by setting $Y[p]$), then the Y variable stays set until p resets it. This is important to ensure that two processes to not concurrently hold the name at the same building block. Unfortunately, this approach necessitates reading all N Y-bits in order to determine whether the Y variable is set. This is the source of the N factor in the $\Theta(Nk)$ time complexity of the algorithm in Figure 7.5. In contrast, the algorithm presented in this section uses at most $k+1$ Y-components per building block. The difficult part of our algorithm is in ensuring that if process p sets the Y variable, then it stays set until p resets it, and that if all processes that have set the Y variable have since reset it, then the Y variable is no longer set. The latter property is important to ensure that a process does not go right from a building block unless some other process is still accessing that building block. If this property is violated, then it is possible for more than one process to reach the same building block $k-1$ steps from the origin of the grid, thereby acquiring the same name.

These properties are ensured through the use of a new technique for setting and resetting the components of Y. In the loop at lines 3 and 4, process p tries to set the Y

variable of building block $(p.i, p.j)$ by assigning p to every $Y[p.i, p.j, p.h]$, where $p.h$ ranges over 0 to $k - p.i - p.j$. Before setting each Y-component, p first checks $X[p.i, p.j]$. If $X[p.i, p.j] \neq p$, then p stops writing Y-components for this building block, resets those it has set (lines 5 and 6), and then moves down to the next building block (line 7).

Observe that each Y component written by p could subsequently be overwritten by another process. Thus, there is a risk that the Y variable does not stay set while p is accessing the building block. However, in Appendix C.1, we show that if p successfully writes all Y-components of a building block, then *some* component stays set until p resets it. To see why this is so, note that, before p does its last write to Y in building block (r, c) (line 4), p first checks that $X[r, c]$ still contains p (line 3). Thus, $X[r, c]$ holds continuously while p writes all of the Y-components (with the possible exception of the last). If some other process q either resets (line 6 or 11) or writes its own identifier (line 4) into one of the Y-components, then q previously either reads $X[r, c] \neq p$ or reads q from that Y component. In either case, it must have done so before p wrote that Y component. This implies that process q can only "corrupt" one of the Y-components p writes. In Appendix C.1, we inductively show that at most $k - r - c - 1$ processes other than p concurrently access building block (r, c). Thus, because p writes $k - r - c$ components of Y *before* its last check of $X[r, c]$, one of the components p sets remains set to p. Also, before process p leaves a building block (either to go to the next one, or because it releases its name), p clears all Y-components that contain p (lines 5 to 6 and lines 10 to 11). Thus, if all processes leave building block (r, c) then the Y variable for that building block is no longer set. These properties capture the essence of the formal correctness proof, which is presented in Appendix C.1.

Theorem 21: Using read and write, wait-free, long-lived $(k(k+1)/2)$-renaming can be implemented so that the worst-case time complexity of acquiring and releasing a name once is $\Theta(k^2)$, and the space complexity is $\Theta(k^3)$. □

A fast, long-lived renaming algorithm that yields a name space whose size is independent of N can be combined with any long-lived renaming algorithm — fast or not — to further reduce the size of the name space. This is achieved by each process first accessing the fast, long-lived renaming algorithm to acquire a name, and then using that name as its process identifier in another long-lived renaming algorithm. In particular, by combining our fast, long-lived renaming algorithm with the (non-fast) ℓ-assignment algorithm (with $\ell = 2k-1$) of Burns and Peterson [26], fast, long-lived renaming can be achieved with a name space of size $2k-1$. As explained earlier, if at most k processes concurrently access Burns and Peterson's algorithm, then their algorithm is wait-free. The worst-case time complexity of acquiring and releasing a name once is $\Theta(Nk^2)$ [80]. Thus, we have the following result. By results of Burns and Peterson [26], and of Herlihy and Shavit [46], this algorithm is optimal with respect to the size of the name space.

Theorem 22: Using read and write, wait-free, long-lived $(2k-1)$-renaming can be implemented so that the worst-case time complexity of acquiring and releasing a name once is $\Theta(k^4)$, and the space complexity is $\Theta(k^4)$. □

7.4 Long-Lived Renaming using Read-Modify-Writes

In this section, we present three wait-free, long-lived renaming algorithms and one lock-free, long-lived algorithm. By using read-modify-write operations, these algorithms significantly improve upon the performance of the algorithms in the previous section. Furthermore, these algorithms yield a name space of size k, which is clearly optimal (the lower bound results of Herlihy and Shavit [46] do not apply to algorithms that employ read-modify-write operations).

The first algorithm uses *set_first_zero* and *clr_bit* to access shared, b-bit variables and has time complexity $\Theta(k/b)$. As discussed earlier, these operations can be implemented, for example, using operations available on the BBN TC2000 [20]. The second algorithm in this section has time complexity $\Theta(\log k)$ — a significant improvement over the first algorithm. To achieve this improvement, this algorithm uses the *bounded_decrement* operation. We then describe how the techniques from these two algorithms can be combined to obtain an algorithm whose time complexity is better than that of either algorithm.

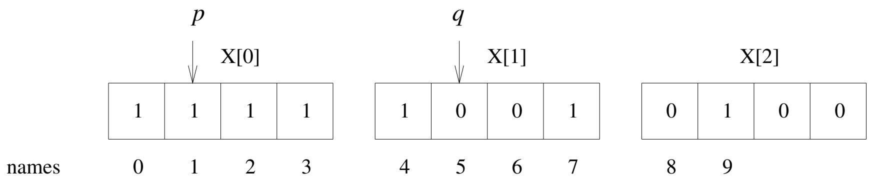
We do not know of any systems that provide *bounded_decrement* as a primitive operation. However, at the end of this section, we discuss how the *bounded_decrement* operation can be approximated in a lock-free manner using the commonly-available *fetch_and_add* operation. We show how this approximation can be used to provide a lock-free algorithm for long-lived k-renaming.

```
shared variable X : array[0..⌈k/b⌉ − 1] of array[0..b − 1] of boolean   /* b-bit "segments" of names */
initially (∀i, j : 0 ≤ i ≤ ⌈k/b⌉ − 1 ∧ 0 ≤ j < b :: X[i][j] = false)

process p                                                              /* 0 ≤ p < N */
private variable h : 0..⌈k/b⌉ − 1; v : 0..b; name : 0..k − 1
initially h = 0

    while true do
0:     Remainder Section;
       h, v := 0, b;                          /* Initialize h and v after remainder section */
       while v = b do                                            /* Loop until a bit is set */
1:        if (∃n : 0 ≤ n < b :: ¬X[h][n]) then  /* set_first_zero operation, as defined in Section 7.1 */
             m := (min n : 0 ≤ n < b :: ¬X[h][n]);  X[h][m], v := true, m
          else
             v := b
          fi;
          if v = b then h := h + 1 fi
       od;
2:     name := bh + v;                                                  /* Calculate name */
       Working Section;
3:     X[h][v] := false                                        /* Clear the bit that was set */
    od
```

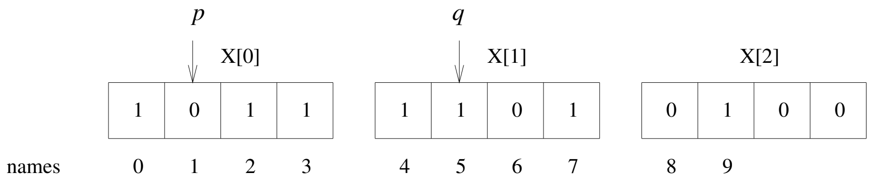
Figure 7.7: Long-lived k-renaming using *set_first_zero* and *clear_bit*.

7.4.1 Long-Lived Renaming using *set_first_zero* and *clr_bit*

Our first long-lived k-renaming algorithm employs the *set_first_zero* and *clr_bit* operations. The algorithm is shown in Figure 7.7. For clarity, we have explicitly used the definitions of *set_first_zero* (statement 1) and *clr_bit* (statement 3). In order to acquire a name, a process tests each name in order. Using the *set_first_zero* operation on b-bit variables, up to b names can be tested in one atomic shared variable access. If $k \leq b$, this results in a long-lived renaming algorithm that acquires a name with just one shared variable access. If $k > b$, then "segments" of size b of the name space are tested in each access. To release a name, a process clears the bit that was set by that process when the name was acquired. An example is shown in Figure 7.8 for $b = 4$ and $k = 10$. In this figure, process p releases name 1 by executing $clr\_bit(X[0], 1)$ and process q acquires name 5 by



(a) In this state, $p@3 \ \wedge \ p.h = 1 \ \wedge \ p.v = 1$ holds, so p is about to execute $clr\_bit(X[0], 1)$, thereby releasing name 1. For process q, $q@1 \ \wedge \ q.h = 1$ holds, so q is about to execute $set\_first\_zero(X[1])$. As $X[1][1]$ is the first clear bit in $X[1]$, $q.1$ will establish $q@2 \ \wedge \ q.h = 1 \ \wedge \ q.v = 1$, and will therefore acquire name 5.



(b) Process p has released name 1 and process q has acquired name 5.

Figure 7.8: Example steps of the k-renaming algorithm shown in Figure 7.7 for $b = 4$ and $k = 10$.

executing $set\_first\_zero(X[1])$.

Because each process tests the available names in segments, and because processes may release and acquire names concurrently, it may seem possible for a process to reach the last segment when none of the names in that segment are available. In Appendix C.2, we show that this is in fact impossible and that each process acquires a distinct name from $\{0, ..., k-1\}$ after at most $\lceil k/b \rceil$ shared variable accesses (see (I151) and (I152)). Releasing a name requires one shared variable access. Thus, the algorithm shown in Figure 7.7 yields the following result.

Theorem 23: Using *set_first_zero* and *clr_bit* on b-bit variables, wait-free, long-lived k-renaming can be implemented so that the worst-case time complexity of acquiring and releasing a name once is $\Theta(k/b)$. □

As discussed earlier, when $b = 1$, the *set_first_zero* and *clr_bit* operations are equivalent to the *test_and_set* and *write* operations, respectively. Thus, we have the following.

Corollary: Using *test_and_set* and *write*, wait-free, long-lived k-renaming can be implemented with time complexity $\Theta(k)$. □

7.4.2 Long-Lived Renaming using *bounded_decrement* and *fetch_and_add*

In this section, we present a long-lived k-renaming algorithm that employs the *bounded_decrement* and *fetch_and_add* operations. In this algorithm, shown in Figure 7.9, the *bounded_decrement* operation is used to separate processes into two groups *left* and *right*. The right group contains at most $\lceil k/2 \rceil$ processes and the left group contains at most $\lfloor k/2 \rfloor$ processes. This is achieved by initializing a shared variable X to $\lceil k/2 \rceil$, and having each process perform a *bounded_decrement* operation on X. Processes that receive positive return values join the right group, and processes that receive zero join the left group. To leave the right group, a process increments X. To leave the left group, no shared variables are updated.

It might seem possible to implement this "splitting" mechanism by having a process join the left group if and only if it receives a nonpositive return value from a normal *fetch_and_add*$(X, -1)$ operation. However, because processes must be able to repeatedly join and leave the groups, the normal *fetch_and_add* operation is not suitable for this "splitting" mechanism. If X is decremented below zero, then it is possible for too many processes to be in the left group at once. To see this, suppose that all k processes decrement X. Thus, $\lceil k/2 \rceil$ processes receive positive return values, and therefore join the right group, and

$\lfloor k/2 \rfloor$ processes receive non-positive return values, and therefore join the left group. Now, $X = -\lfloor k/2 \rfloor$. If a process leaves the right group by incrementing X, and then decrements X as the result of a subsequent attempt to acquire a name, then that process receives a non-positive return value, and thus joins the left group. Repeating this for each process in the right group, it is possible for all processes to be in the left group simultaneously. The *bounded_decrement* operation prevents this by ensuring that X does not become negative.

The algorithm employs an instance of long-lived $\lceil k/2 \rceil$-renaming for the right group, and an instance of long-lived $\lfloor k/2 \rfloor$-renaming for the left group, which are inductively assumed to be correct. For notational convenience, we assume that a name is acquired from the left instance by calling *Getname_left* and released by calling *Putname_left*; similarly for the right instance. (These functions are easy to implement given the inductively-assumed instances.) The algorithm that results from "unfolding" this inductively-defined algorithm forms a tree. To acquire a name, a process goes down a path in this tree from the root to a leaf. As the processes progress down the tree, the number of processes that can simultaneously go down the same path is halved at each level. When this number becomes one, a name can be assigned. Thus, the time complexity of acquiring a name is $\Theta(\log k)$. To release a name, a process retraces the path it took through the tree in reverse order, incrementing X at any node at which it received a positive return value.

Note that, with b-bit variables, if $b < \log_2 \lceil k/2 \rceil$, then X cannot be initialized to $\lceil k/2 \rceil$, so this algorithm cannot be implemented. However, in any practical setting, this will not be the case. In Appendix C.3, we prove the following result.

```
shared variable X : 0..⌈k/2⌉                    /* Counter of names available on right */
initially X = ⌈k/2⌉

process p                                        /* 0 ≤ p < N */
private variable side : {left, right}

   while true do
      Remainder Section;
      if bounded_decrement(X) > 0 then           /* Ensure at most ⌈k/2⌉ (⌊k/2⌋) access right (left) */
         side, name := right, Getname_right()    /* Get name from right instance */
      else
         side, name := left, ⌈k/2⌉ + Getname_left()   /* Get name from left instance */
      fi;
      Working Section;
      if side = right then
         Putname_right(name);                    /* Return name to right instance */
         fetch_and_add(X, 1)                     /* Increment counter again */
      else
         Putname_left(name − ⌈k/2⌉)              /* Return name to left instance */
      fi
   od
```

Figure 7.9: k-renaming using *bounded_decrement*. *Getname_left* and *Putname_left* are inductively assumed to implement long-lived $\lfloor k/2 \rfloor$-renaming. Similarly, *Getname_right* and *Putname_right* are inductively assumed to implement long-lived $\lceil k/2 \rceil$-renaming.

Theorem 24: Using b-bit variables and *bounded_decrement* and *fetch_and_add*, wait-free, long-lived k-renaming can be implemented so that the worst-case time complexity of acquiring and releasing a name once is $\Theta(\log k)$ for $k \leq 2(2^b - 1)$. $\square$

Note that if the *set_first_zero* and *clr_bit* operations are available, then it is unnecessary to completely "unfold" the tree algorithm described above. If the tree is deep enough that at most b processes can reach a leaf, then by Theorem 23, a name can be assigned with one more shared access. This amounts to "chopping off" the bottom $\lfloor \log_2 b \rfloor$ levels of the tree. The time complexity of the resulting algorithm is $\Theta(\log k - \log b) = \Theta(\log(k/b))$. Thus, using all the operations employed by the first two algorithms, we can improve on the time complexity of both. The following result is proved in Appendix C.3.

```
shared variable X : −⌊k/2⌋..⌈k/2⌉                /* Counter of names available on right */
initially X = ⌈k/2⌉

process p                                        /* 0 ≤ p < N */
private variable side : {left, right, none}

   while true do
      Remainder Section;
      side := none;
      while side = none do
         if fetch_and_add(X, −1) > 0 then side := right
         else if fetch_and_add(X, 1) < 0 then side := left fi
         fi
      od;
      if side = right then
            name := Getname_right()              /* Get name from right instance */
      else
         name := ⌈k/2⌉ + Getname_left()          /* Get name from left instance */
      fi;
      Working Section;
      if side = right then
         Putname_right(name);                    /* Return name to right instance */
         fetch_and_add(X, 1)                     /* Increment counter again */
      else
         Putname_left(name − ⌈k/2⌉)              /* Return name to left instance */
      fi
   od
```

Figure 7.10: Lock-free k-renaming using *fetch_and_add*.

Theorem 25: Using b-bit variables and *set_first_zero*, *clear_bit*, *bounded_decrement*, and *fetch_and_add*, wait-free, long-lived k-renaming can be implemented so that the worst-case time complexity of acquiring and releasing a name once is $\Theta(logk/b)$ for $1 \leq k \leq 2(2^b - 1)$.$\square$

7.4.3 Lock-Free, Long-Lived Renaming using *fetch_and_add*

The k-renaming algorithm presented in Figure 7.9 is the basis of our fastest wait-free k-renaming solutions, as shown by Theorems 24 and 25. Unfortunately, the *bounded_decrement* operation employed by that algorithm is not widely available. While the *bounded_decrement* operation is similar to the well-known *fetch_and_add* operation, we have been unable to design an efficient wait-free implementation of the former using the latter. We have, however, designed a lock-free k-renaming algorithm that is based on the idea of *bounded_decrement*. The algorithm is presented in Figure 7.10. The *fetch_and_add* operation is used to approximate the *bounded_decrement* operation in such a way that it ensures that at most $\lceil k/2 \rceil$ processes access the right instance of $\lceil k/2 \rceil$-renaming, and similarly for the left instance.

The split is achieved by having processes that obtain positive values from X go right, and processes that obtain non-positive values go left (see statements 1 and 2 in Figure 7.10). However, a process, say p, that decrements the counter X below zero "compensates" by incrementing X again before proceeding left. If p detects that X becomes positive again before this compensation is made, then it is possible that some other process has incremented X and joined the left group. In this case, there is a risk that process p should in fact go right, rather than left. In this case, process p restarts the loop.

The algorithm is lock-free because in order for a process to repeat the loop at statements 1 and 2, some other process must modify X between the execution of statements 1 and 2. In Appendix C.4, we show that if this happens repeatedly, then eventually some process makes progress. Thus, we have the following result.

Theorem 26: Using b-bit variables and *fetch_and_add*, lock-free, long-lived k-renaming can be implemented so that the worst-case, contention-free time complexity of acquiring and releasing a name once is $\Theta(\log k)$ for $k \leq 2(2^b - 1)$. $\square$

Chapter 8

Conclusions

In this chapter, we first give a brief summary of the results presented in this dissertation, and discuss possible directions for future research on these results. We then make some concluding remarks about the lessons learned from this research.

8.1 Summary

In Chapter 4, we presented several results that allow the algorithms presented in Chapters 5 and 6, as well as other algorithms found in the literature, to be applied with greater flexibility. These results also have the potential to ease the design of future algorithms, by giving researchers greater flexibility in choosing synchronization instructions on which to base their work. Specifically, we present constant-time, wait-free implementations of compare-and-swap using LL/SC and vice versa; we give an implementation that obviates the need to deal with "spurious" SC failures; and we give an implementation of weak-LL, VL, and SC that removes the limitation on the size of the variables accessed that is imposed by most hardware implementations of these instructions. All of these implementations have optimal time complexity, but it may be possible to reduce their space complexity.

The constructions in Chapter 5 improve the space and time efficiency of 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 assignment statements used in conventional programming. Nonetheless, as demonstrated by Figure 5.3, 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.

Our constructions do not exploit parallel execution of operations, even if the operations access disjoint sets of blocks. We would like to extend our constructions to take advantage of such parallel execution where possible. For example, in our shared queue implementations, an *enqueue* operation might unnecessarily interfere with a *dequeue* operation. In [9], we addressed similar concerns when implementing wait-free operations on multiple objects.

In Chapter 6, we presented a technique for improving the time and space efficiency of wait-free, universal constructions, as well as the space efficiency of lock-free ones. This technique is based on efficient k-assignment algorithms. To facilitate this approach, we presented several shared-memory algorithms for k-exclusion and k-assignment in which all process blocking is achieved through the use of "local-spin" busy waiting. These algorithms are designed to minimize interconnect traffic associated with busy waiting on cache-coherent and distributed shared-memory multiprocessors. Furthermore, they are based on commonly-available synchronization primitives, are fast in the absence of contention, and exhibit scalable performance as contention rises. In contrast, all prior k-exclusion algorithms either require unrealistic atomic operations, or have unacceptably high time complexity. To our knowledge, these algorithms are the first local-spin synchronization algorithms to tolerate process failures. We also presented performance studies conducted on cache-coherent and distributed shared-memory multiprocessors involving shared object implementations that are obtained by combining our k-assignment algorithms with wait-free object implementations. These studies show that, in multiprogrammed systems, the implementations we consider often outperform both wait-free and spin-lock-based object implementations. These results validate our claims that our k-exclusion and k-assignment algorithms are fast and scalable.

It is interesting to note that we had to use quite old multiprocessors for our performance experiments and, in the case of the Sequent Symmetry, we had to simulate multiprogramming. This is because we were unable to find more modern shared-memory multiprocessors that robustly support multiprogramming. For example, we found the multiprogramming support on the KSR to be extremely unreliable: our experiments would frequently hang for several hours before completing. Also, we tried to perform multiprogramming experiments on a Convex Exemplar, but found that an operating system bug prevented applications from creating more threads than the available number of processors. (We reported the above-mentioned bug to Convex and they have since corrected it.) We feel that this lack of support for multiprogramming is largely a result of the fact that multiprogramming is rarely used in multiprocessor applications, and that this in turn is a result of the difficulty of achieving efficient interprocess synchronization under multiprogramming. We hope that our work will help to alleviate this difficulty.

It would be interesting to try to improve upon our results by developing k-exclusion algorithms for which performance under contention is completely independent of N. Ideally, we would like for such algorithms to have performance that approaches that of the fastest spin-lock algorithms when k approaches 1.

In Chapter 7, we considered the classic one-time renaming problem, as well as a more general problem called long-lived renaming, in which processes can release names as well as acquire names. We provided several solutions to this problem, including some that employ only read and write operations. We also presented a new one-time renaming algorithm, which improves on previous read/write renaming algorithms in that its time complexity is independent of the size of the original name space.

Our renaming algorithms exhibit a trade-off between time complexity, name space size, and the availability of primitives used. Most of our wait-free algorithms have the desirable property that time complexity is proportional to contention. Thus, if fewer than k processes concurrently use a particular renaming algorithm, then the worst-case time complexity of acquiring and releasing a name is lower than the time complexity stated in our theorems. This is an important practical advantage because contention should be low in most well-designed applications [65].

Our study of read/write algorithms for long-lived renaming has culminated in an algorithm that is fast and achieves an optimal name space size of $2k-1$, while having time complexity that is independent of the size of the original name space. However, because this result is achieved by combining two algorithms, it has quite high time complexity ($\Theta(k^4)$). It would be interesting to see whether this can be improved upon by a direct solution.

Our most efficient wait-free, long-lived renaming algorithm uses a *bounded_decrement* operation. Although this operation is similar to the standard *fetch_and_add* operation, we have been unable to design an efficient wait-free implementation of the former using the latter. We have, however, designed an efficient lock-free implementation of k-renaming based on this idea. In this implementation, a process can only be delayed by a very unlikely sequence of events. We believe this implementation will perform well in practice. It remains to be seen whether *fetch_and_add* can be used to implement wait-free, long-lived renaming with sub-linear time complexity.

8.2 Conclusions and Future Research

In the course of this research, we have learned several important lessons about implementing shared objects in shared-memory multiprocessors.

- The performance of universal constructions is less dependent on the choice of synchronization primitive than was previously thought. In particular, we have shown that constructions that are based on the LL/SC instruction pair can be modified so that they use compare-and-swap with minimal additional time overhead. Such a construction might be useful to architects who are deciding which hardware instructions to support. (While the gap between compare-and-swap and LL/SC has been shown to be substantially smaller than previously thought, the construction of LL/SC using compare-and-swap does have some computational overhead, albeit constant, and also a non-trivial space overhead. As discussed in Section 4.1.1, this space overhead can be greatly reduced for many applications by using unbounded counters. Nonetheless, the LL/SC pair is probably still preferable to algorithm designers.) On the subject of hardware instructions, it is interesting to note that fetch-and-add has proved useful in several settings, but no efficient, wait-free implementation of fetch-and-add from either compare-and-swap or LL/SC is known (although lock-free implementations are trivial). Results of Anderson [12] and Cypher [28] imply that no constant-time, wait-free implementation of fetch-and-add from compare-and-swap or LL/SC exists. However, the lower bound implied by these results is quite low, leaving a significant gap between the lower bound and the best known implementation. Resolving this gap with a significantly higher bound would suggest that architectures that provide only CAS or LL/SC and not fetch-and-add are not sufficient to support efficient, wait-free (or even starvation-free) algorithms that are based on fetch-and-add.

- There are practical benefits to be derived from the ability of an object implementation to withstand process failures (even if there are no failures). This is because withstanding process failures implies withstanding arbitrary process delays, which are common and unpredictable in modern multiprocessors. As seen in Chapter 6, wait-free implementations can outperform lock-based implementations under multiprogramming. However, because there is necessarily an overhead associated with providing the advantages of wait-freedom, there will always be a tradeoff in deciding whether to use a wait-free implementation. This decision is likely to be dependent on the target machine, the operating system, the application and other applications that run on the machine, the frequency of object accesses, the expected contention for objects, distribution of process delays, and numerous other factors. Before these decisions can be easily made, and wait-free (or other non-blocking) object implementations come into common usage, two research directions must be pursued. First, while we have substantially improved on the overhead of previous wait-free universal constructions, there is no reason to believe that our constructions are optimal. Reducing these overheads further will allow the benefits of wait-freedom to be realized in a wider variety of settings. Second, case studies and performance studies are needed in order to accurately characterize the applications and environments in which wait-free object implementations will provide performance benefits.

- The techniques presented in Chapter 6 allow us to construct lower overhead implementations that provide all the benefits of wait-free implementations while contention remains within expected limits. Good performance can also be achieved using lock-free constructions. However, they must be used in conjunction with carefully-tuned backoff mechanisms to ensure that they provide good performance under contention. This is particularly problematic in applications that have different object access behaviors at different times, because it requires the backoff mechanism to be tuned dynamically. Wait-free implementations whose time complexity depends on contention, as proposed by Afek, Dauber, and Touitou [2], have the potential overcome these problems. However, efficient constructions with this property remain to be seen.

- Existing wait-free universal constructions (including ours) do not benefit from parallel execution of operations. This could prove to be a severe disadvantage, because it limits scalability. Therefore, designing universal constructions that do exploit parallelism is an important research direction. We have made some progress towards this goal elsewhere [9] by designing constructions for implementing multiple objects and supporting multi-object operations that execute in parallel where possible. Drawing on the techniques presented in [9] and in Chapter 5, it should be possible to implement general, wait-free "transactions" (arbitrary operations) that execute in parallel if they do not overlap.

Appendix A

Correctness Proofs for Algorithms in Chapter 5

In this appendix, we provide a formal correctness proof for the wait-free construction for large objects presented in Section 5.2.

A.1 Correctness Proof for Algorithm in Figures 5.5 and 5.6

In order to model the interaction of each process with our construction, we assume the existence of a statement 0 (not shown in Figures 5.5 and 5.6) that repeatedly calls *WF_Op* with appropriate parameters. When process p returns from the *WF_Op* procedure (line 50), p's program counter is set to 0 in preparation for p's next operation invocation. The *Simulate_Op* procedure (line 19) models user-supplied operations. This is achieved by using a nondeterministic choice to produce a sequence of read and write operations to the implemented BS-word array, followed by the return of a nondeterministic value. We use the *select* function, which takes a type parameter and returns an arbitrary value of that type, to achieve this nondeterminism.

The algorithm in Figures 5.5 and 5.6 is augmented with several auxiliary variables and functions, which are used to construct a total order over operation invocations that is consistent with the sequential semantics of the implemented object. The linearizability proof shows that each operation invocation is placed into this total order exactly once, that this occurs during the execution of that invocation (i.e., the constructed total order extends the partial order over invocations), and that each invocation returns the correct return value with respect to the total order constructed.

We now explain how the total order over operations is constructed. While performing operations, each process p maintains an auxiliary variable *p.auxcopy* which records the changes made to p's local view of the implemented object. (Observe that, for each write operation invoked by an operation performed by process p, an equivalent write operation is performed on *p.auxcopy* at line 18.) Process p records the operations it performs in an auxiliary list *p.hlplst* (line 25), which contains one tuple for each operation performed. Each tuple contains the identity of the process that invoked the operation, the operation itself, the parameters passed to that operation, and the value that would be returned by a sequential implementation of that operation, given the previous object state in *p.auxcopy*. Each time process p performs a successful SC operation (line 11), the operations in *p.hlplst* are linearized (i.e., added to the constructed total order) in the order that they appear in *p.hlplst*. This is recorded by updating the auxiliary variable *AuxObj*, which contains the current value of the implemented object. This is achieved by means of the *ApplyAll* function, which is defined below.

Definition: The *ApplyAll* function takes two parameters: a current object state (an array of BS words) and a list of tuples. *ApplyAll* scans the list in order and, for each tuple $(q, op, pars, ret)$, sequentially applies the operation op, with parameters $pars$ to the current object state. *ApplyAll* returns the state of the object that results from all of these operation applications. (Note that this function is used only to facilitate the proof: it is not actually implemented.) □

Observe that, when p executes a LL at line 1, p also copies *Auxcopy* to *p.auxcopy*. Thus, because *Auxcopy* does not change between p's LL and p's successful SC (if it did change, then p's SC would fail), the new value written to *Auxcopy* when p performs its successful SC correctly reflects the sequential execution of the operations in *p.hlplst*. Thus, the return values of the operations in the total order constructed are consistent with the sequential semantics of those operations.

Because only one variable (X) is accessed using the LL, VL, and SC operations, we use the following simple axioms to model the behavior of these operations for this algorithm. As can be seen from Figure 3.1, these are consistent with the semantics of LL, VL, and SC.

Axiom 1: For any process p, the execution of $LL(X)$ by process p establishes $still\_valid(p)$. No other statement establishes $still\_valid(p)$.

Axiom 2: If $\neg still\_valid(p)$ holds, then an execution of $SC(X, v)$ by process p does not modify X and returns *false*.

Axiom 3: If *still_valid*(p) holds, then an execution of $SC(X, v)$ by process p writes v to X and returns *true*. $SC(X, v)$ also establishes $\neg$*still_valid*(q) for all processes q in this case. No other statement falsifies *still_valid*(p).

Axiom 4: An execution of $VL(X)$ by process p returns *true* if *still_valid*(p) holds and *false* otherwise.

There are many dependencies between the properties in the correctness proof for the algorithm in Figures 5.5 and 5.6. In particular, we sometimes assume a property that has not yet been proved (including the property we are currently proving). It is important to note that this apparent circularity does not render our proof incorrect. This is because, in such cases, we assume only that the property holds *before* [1] each statement execution. Thus, we inductively prove the *conjunction* of all of the properties: we show for each property that it holds initially, and that, assuming all properties hold in one state, the given property holds in every subsequent state.

The preliminaries discussed in Chapter 3 apply to the correctness proof presented below. In particular, recall that p, q, and r range over $\{0 \ldots N-1\}$, and that all other unbound variables are assumed to be universally quantified. Figures A.1 and A.2 contain some definitions that are used in the correctness proof presented below.

The following invariants are trivial, and are therefore stated without proof. In particular, the proof of (I8) uses (I7); (I13) uses (I5), (I6), and (I12); (I15) uses (I14); (I22) uses (I21); (I23) uses (I22); (I24) uses (I23); (I25) uses (I24); and (I29) uses (I22).

[1] For brevity, we simply refer to the property in such cases, rather than explicitly stating that we are assuming only that the property holds before each statement execution. Nonetheless, we *never* assume that a later property (or the property being proved) holds *after* a statement execution.

$$\textbf{invariant } q@\{7..25, 33..46\} \Rightarrow q.rb = q.ptrs.ret \tag{I5}$$

$$\textbf{invariant } q@11 \Rightarrow BUF[q, q.side].ret = q.ptrs.ret \tag{I6}$$

$$\textbf{invariant } q@14 \Rightarrow 0 \leq q.addr < B * S \tag{I7}$$

$$\textbf{invariant } q@\{15..18\} \Rightarrow 0 \leq q.blkidx < B \ \wedge \ q.blkidx = q.addr \textbf{ div } S \tag{I8}$$

$$\textbf{invariant } q@\{15..18\} \Rightarrow q.addr = (q.addr \textbf{ div } S) * S + (q.addr \textbf{ mod } S) \tag{I9}$$

$$\textbf{invariant } q@\{15..17\} \Rightarrow \neg q.dirty[q.blkidx] \tag{I10}$$

$$\textbf{invariant } q@18 \Rightarrow q.dirty[q.blkidx] \tag{I11}$$

$$\textbf{invariant } q@\{45, 7..11\} \Rightarrow LAST[q] = q.rb \tag{I12}$$

$$\textbf{invariant } q@11 \Rightarrow BUF[q, q.side].ret = LAST[q] \tag{I13}$$

$$\textbf{invariant } q@8 \Rightarrow (\forall m : 0 \leq m < q.i :: BUF[q, q.side].blks[m] = q.ptrs.blks[m]) \tag{I14}$$

$$\textbf{invariant } q@\{9..11\} \Rightarrow BUF[q, q.side].blks = q.ptrs.blks \tag{I15}$$

$$\textbf{invariant } p.bit = ANC[p].bit \tag{I16}$$

$$\textbf{invariant } q@21 \ \wedge \ q.pr = q \Rightarrow q.match = ANC[q].bit \tag{I17}$$

$$\textbf{invariant } q@\{12..25\} \Rightarrow q.pr = q.try \tag{I18}$$

$$\textbf{invariant } q@\{20..21\} \ \wedge \ q.pr = q \Rightarrow q.hlplst = \{\} \tag{I19}$$

$$\textbf{invariant } q@\{12..25\} \ \wedge \ q.from = 40 \Rightarrow q.hlplst = \{\} \tag{I20}$$

$$\textbf{invariant } p.side \in \{0, 1\} \tag{I21}$$

$$\textbf{invariant } 0 \leq X.pid < N \ \wedge \ X.tag \in \{0, 1\} \tag{I22}$$

$$
\begin{aligned}
RV(p) &\equiv RET[BUF[X.pid, X.tag].ret][p].val\\
AV(p) &\equiv RET[BUF[X.pid, X.tag].ret][p].applied\\
CV(p) &\equiv RET[BUF[X.pid, X.tag].ret][p].copied\\
NORM(p) &\equiv AV(p) = ANC[p].bit \ \wedge \ CV(p) = ANC[p].bit\\
ST(p) &\equiv (AV(p)+1) \ \mathbf{mod} \ 3 = ANC[p].bit \ \wedge \ (CV(p)+1) \ \mathbf{mod} \ 3 = ANC[p].bit\\
APP(p) &\equiv AV(p) = ANC[p].bit \ \wedge \ (CV(p)+1) \ \mathbf{mod} \ 3 = ANC[p].bit\\
CPD(q,v,p) &\equiv (\neg q@25 \ \vee \ q.pr \neq p) \ \wedge \ RET[v][p].val = RV(p) \ \wedge\\
&\quad\ RET[v][p].applied = AV(p) \ \wedge \ RET[v][p].copied = RET[v][p].applied\\
IBS(q,v,p) &\equiv q@25 \ \wedge \ q.pr = p \ \wedge \ RET[v][p].val = q.rv \ \wedge\\
&\quad\ RET[v][p].applied = AV(p) \ \wedge \ RET[v][p].copied = RET[v][p].applied\\
APD(v,p,val) &\equiv RET[v][p].val = val \ \wedge \ ST(p) \ \wedge \ ANC[p].bit = RET[v][p].applied \ \wedge\\
&\quad\ ANC[p].bit = (RET[v][p].copied + 1) \ \mathbf{mod} \ 3\\
CRV(q,v,p) &\equiv (\neg(\exists \ tup :: tup \in q.hlplst \ \wedge \ tup.pid = p) \ \wedge\\
&\qquad\qquad\qquad\qquad (CPD(q,v,p) \ \vee \ IBS(q,v,p))) \ \vee\\
&\quad\ (\exists \ tup :: tup \in q.hlplst \ \wedge \ tup.pid = p \ \wedge \ tup.op = ANC[p].op \ \wedge\\
&\qquad\qquad\qquad tup.pars = ANC[p].pars \ \wedge \ APD(v,p,tup.val))\\
ACRV(q,v) &\equiv (\forall p :: CRV(q,v,p))
\end{aligned}
$$

Figure A.1: Definitions used in the correctness proof for the algorithm in Figures 5.5 and 5.6.

$$
\begin{aligned}
&EQB(n, m, Aux) \equiv (\forall x : 0 \le x < S :: Aux[m * S + x] = BLKS[n][x]) \\
&EQ(Arr, Aux) \equiv (\forall m : 0 \le m < B :: EQB(Arr[m], m, Aux)) \\
&HAS(v, f, t, n) \;\equiv (\exists h : f \le h < t :: v[h] = n) \\
&HL(p, n) \qquad\quad \equiv HAS(p.oldlst, 0, p.dcnt, n) \;\vee\; HAS(p.copy, p.dcnt, M, n) \\
&UNIQ(v, m) \quad\;\; \equiv (\forall h, k : 0 \le h < m \;\wedge\; 0 \le k < m \;\wedge\; h \ne k :: v[h] \ne v[k]) \\
&STS(p, n) \qquad \equiv \; (RET[n][p].copied + 1) \textbf{ mod } 3 = ANC[p].bit \\
&WND(q, p) \quad\;\; \equiv \; \neg q@\{12..25, 38..44\} \vee \\
&\qquad\qquad\qquad\quad (q@\{38..39\} \;\wedge\; (RET[q.rb][p].applied + 1) \textbf{ mod } 3 = ANC[p].bit) \vee \\
&\qquad\qquad\qquad\quad (q@\{12..25, 40..44\} \;\wedge\; STS(p, q.rb)) \\
&SRV(p, n) \qquad \equiv \; RET[n][p].val = RV(p) \\
&DND(q, p) \quad\;\; \equiv \; \neg q@\{12..25, 38..44\} \vee \\
&\qquad\qquad\qquad\quad (SRV(p, q.rb) \;\wedge\; RET[q.rb][p].applied = ANC[p].bit \;\wedge \\
&\qquad\qquad\qquad\quad (q@\{38..44\} \;\vee\; (q@\{12..25\} \;\wedge\; q.pr \ne p) \;\vee\; (q@20 \;\wedge\; q.pr = p) \;\vee \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (q@21 \;\wedge\; q.pr = p \;\wedge\; q.match = ANC[p].bit)))
\end{aligned}
$$

Figure A.2: Definitions used in the correctness proof for the algorithm in Figures 5.5 and 5.6 (continued from Figure A.1).

invariant $0 \le p.curr.pid < N \;\wedge\; p.curr.tag \in \{0, 1\}$ (I23)

invariant $0 \le p.ptrs.help < N \;\wedge$

$$0 \le BUF[p, 0].help < N \;\wedge\; 0 \le BUF[p, 1].help < N \qquad \text{(I24)}$$

invariant $0 \le p.try < N$ (I25)

invariant $p@\{2..5\} \;\wedge\; still\_valid(p) \Rightarrow p.curr = X$ (I26)

invariant $p@\{33..40\} \Rightarrow p.dcnt = 0$ (I27)

invariant $p@6 \Rightarrow \neg still\_valid(p)$ (I28)

invariant $p@27 \Rightarrow 0 \le p.tmp \le N$ (I29)

invariant $p@\{1..6, 26..27\} \;\wedge\; p.from = 29 \Rightarrow \neg p.done$ (I30)

invariant $NORM(p) \Rightarrow \neg ST(p) \;\wedge\; \neg APP(p)$ (I31)

invariant $ST(p) \Rightarrow \neg NORM(p) \;\wedge\; \neg APP(p)$ (I32)

invariant $APP(p) \Rightarrow \neg NORM(p) \;\wedge\; \neg ST(p)$ (I33)

Correctness proofs are presented below for the remaining properties.

invariant $p@\{15..17\} \Rightarrow 0 \le p.dcnt < M$ (I34)

Proof: Recall that we assume that each sequential operation modifies at most T blocks of the array. Thus, because (I27) implies that process p calls *Apply* (and therefore *Simulate_Op*) only if $p.dcnt + T \le M$ holds, and because $p.dcnt$ is increased by one for each distinct block modified by a sequential operation, it is easy to see that (I34) holds. □

invariant $0 \le p.dcnt \le M$ (I35)

Proof: Initially, $p.dcnt = 0$ holds, so (I35) holds. Only statements $p.17$ and $p.32$ modify $p.dcnt$. By (I34), neither statement falsifies (I35). $\square$

invariant $p@\{2..25, 32..45\} \ \wedge\ \neg still\_valid(p) \Rightarrow X.pid \neq p$ (I36)

Proof: Initially, $p@0$ holds, so (I36) holds. Only statement $p.1$ establishes $p@\{2..25, 32..45\}$, and by Axiom 1, statement $p.1$ does not establish the antecedent. By Axiom 3, only statement $s.11$, where s is any process, establishes $\neg still\_valid(p)$ or modifies X, and by Axiom 2, it does so only if executed when $s@11 \ \wedge\ still\_valid(s)$ holds. If $s = p$, then $p@46$ holds, and therefore the antecedent does not hold after the execution of statement $s.11$. If $s \neq p$, then by Axiom 3, the consequent holds after the execution of $s.11$. $\square$

invariant $\neg p@\{8..11\} \Rightarrow X.pid \neq p \ \vee\ X.tag = p.side$ (I37)

Proof: Initially, the consequent holds, so (I37) holds. Only statement $p.11$ establishes the antecedent. If $p@11 \ \wedge\ \neg still\_valid(p)$ holds before the execution of statement $p.11$, then, by (I36), the consequent holds before the execution of statement $p.11$, and, by Axiom 2, statement $p.11$ does not falsify the consequent. If $p@11 \ \wedge\ still\_valid(p)$ holds before the execution of statement $p.11$, then by Axiom 3, statement $p.11$ establishes the consequent. Only statement $s.11$, where s is any process, modifies X. As shown above, if $s = p$, then the consequent holds after the execution of statement $s.11$. If $s \neq p$, then, by Axioms 2 and 3, if statement $s.11$ modifies X, then $X.pid \neq p$ hold afterwards. $\square$

invariant $p@\{8..11\} \Rightarrow X.pid \neq p \ \vee\ X.tag \neq p.side$ (I38)

Proof: Initially, $p@0$ holds, so (I38) holds. Only statement $p.7$ establishes the antecedent. By (I37), the consequent holds after the execution of statement $p.7$. No statement modifies

$p.side$ while the antecedent holds. Only statement $s.11$, where s is any process, modifies X, and by Axiom 2, it does so only if executed when $s@11 \wedge still\_valid(s)$ holds. If $s = p$, then the antecedent does not hold after the execution of statement $s.11$. If $s \neq p$, then, by Axiom 3, the consequent holds after the execution of statement $s.11$. □

Claim 2: No statement modifies $BUF[p, t]$ while $X = (p, t)$ holds.

Proof: Claim 2 follows directly from the program text and (I38). □

invariant $q@\{5, 32\} \wedge still\_valid(q) \Rightarrow q.ptrs.ret = BUF[X.pid, X.tag].ret$ (I39)

Proof: Initially, $q@0$ holds, so (I39) holds. By Axiom 1, no statement establishes $still\_valid(q)$ while $q@\{5, 32\}$ holds. By (I22), only statement $q.5$ establishes $q@32$. Only statement $q.4$ establishes $q@5$, and, by Axiom 1, statement $q.4$ establishes the antecedent only if executed when $q@4 \wedge still\_valid(q)$ holds. By (I26), statement $q.4$ establishes the consequent in this case.

No statement modifies $q.ptrs.ret$ while the antecedent holds. Thus, by Claim 2, the consequent can be falsified only by modifying X. Only statement $s.11$, where s is any process, modifies X. By Axioms 2 and 3, if $s.11$ modifies X, then it also falsifies $still\_valid(q)$, thereby falsifying the antecedent. □

invariant $q@\{7..25, 33..45\} \wedge still\_valid(q) \Rightarrow q.oldrb = BUF[X.pid, X.tag].ret$ (I40)

Proof: Initially, $q@0$ holds, so (I40) holds. By Axiom 1, only statement $q.32$ establishes the antecedent, and it does so only if executed when $q@32 \wedge still\_valid(q)$ holds. By (I39), statement $q.32$ establishes the consequent in this case.

No statement modifies $q.oldrb$ while the antecedent holds. Therefore, by Claim 2 and Axioms 2 and 3, no statement falsifies the consequent while the antecedent holds. $\square$

invariant $p \neq q \ \wedge \ ((p@46 \ \wedge \ p.oldrb = n) \ \vee \ (\neg p@46 \ \wedge \ p.rb = n)) \Rightarrow$

$$(\neg q@46 \ \vee \ q.oldrb \neq n) \ \wedge \ (q@46 \ \vee \ q.rb \neq n) \ \wedge$$
$$BUF[X.pid, X.tag].ret \neq n \ \wedge$$
$$(\neg still\_valid(q) \ \vee \ q.oldrb \neq n \ \vee \ \neg q@\{7..25, 33..45\}) \tag{I41}$$

Proof: Initially, $p@0 \ \wedge \ BUF[X.pid, X.tag].ret \neq p.rb \ \wedge \ q@0 \ \wedge \ q.rb \neq p.rb$ holds, so (I41) holds. No statement modifies $p.rb$ while $\neg p@46$ holds, and no statement modifies $p.oldrb$ while $p@46$ holds. Only statement $p.46$ establishes $\neg p@46$, and the antecedent holds after the execution of $p.46$ only if $p@46 \wedge p.oldrb = n$ holds before. Thus, statement $p.46$ does not establish the antecedent. Only statement $p.11$ establishes $p@46$. By Axiom 2, the antecedent holds after the execution of statement $p.11$ only if $p@11 \ \wedge \ still\_valid(p) \ \wedge \ p.oldrb = n \ \wedge \ p.rb \neq n$ holds before. By $(\text{I41})^{p,q}_{q,p}$, this implies that the first two conjuncts of the consequent hold before the execution of statement $p.11$ (the last conjunct of the consequent of $(\text{I41})^{p,q}_{q,p}$ does not hold in this case, so neither does the antecedent); statement $p.11$ does not falsify either of them. Also, (I12) and (I13) imply that $BUF[p, p.side].ret \neq n$ holds before the execution of statement $p.11$ in this case. Therefore, by Axiom 3, statement $p.11$ establishes the third conjunct of the consequent. Finally, by Axiom 3, statement $p.11$ establishes $\neg still\_valid(q)$, so the last conjunct of the consequent holds after the execution of $p.11$. We now consider statements that potentially falsify the consequent while the antecedent holds.

Only statement $q.32$ modifies $q.oldrb$, and $\neg q@46$ holds after the execution of statement $q.32$. Only statement $q.11$ establishes $q@46$, and, by Axiom 2, it does so only if executed when $q@11 \ \wedge \ still\_valid(q)$ holds. The last conjunct of the consequent implies that $q.oldrb \neq n$ holds before, and therefore after, the execution of statement $q.11$ in this case.

Only statement $q.46$ can falsify $q@46$ or modify $q.rb$. The first conjunct of the consequent implies that $q.oldrb \neq n$ holds before the execution of statement $q.46$, which implies that $q.rb \neq n$ holds afterwards.

By Claim 2, no statement falsifies the third conjunct of the consequent by modifying BUF. By Axiom 2, X is modified only by statement $s.11$, where s is any process, and only if $s.11$ is executed when $s@11 \ \wedge \ still\_valid(s)$ holds. If $s = p$, then by (I40), $p@46 \ \wedge \ p.oldrb \neq n$ holds after the execution of statement $s.11$ in this case, so the antecedent does not hold. If $s \neq p$, then by $(\text{I41})^q_s$, $s.rb \neq n$ holds before the execution of statement $s.11$, so, by (I12) and (I13), $s.11$ does not falsify the third conjunct of the consequent.

By Axiom 1, only statement $q.1$ establishes $still\_valid(p)$, and $q@2$ holds after the execution of statement $q.1$. Only statement $q.32$ modifies $q.oldrb$ or establishes $q@\{7..25, 33..45\}$. By Axiom 1, if $q@32 \ \wedge \ \neg still\_valid(q)$ holds before the execution of statement $q.32$, then $\neg still\_valid(q)$ also holds afterwards. If $q@32 \ \wedge \ still\_valid(q)$ holds before the execution of statement $q.32$, then, by (I39) and the third conjunct of the consequent, $q.oldrb \neq n$ holds afterwards. $\Box$

Claim 3: Only statement $p.49$ modifies $RET[m][p]$ while $BUF[X.pid, X.tag].ret = m$ holds.

Proof: Claim 3 follows directly from the program text and (I41). $\Box$

Claim 4: $RV(p)$ and $AV(p)$ change only if X is modified.

Proof: By Claim 2, $RV(p)$ and $AV(p)$ do not change as a result of a statement modifying BUF. By Claim 3, $RV(p)$ and $AV(p)$ do not change as a result of a statement modifying RET. □

Claim 5: Statements other than $p.17$, $p.32$, and $p.46$ do not establish or falsify $HL(p, n)$ for any n.

Proof: The claim follows easily from the program text and the definition of HL. □

invariant $0 \leq h < B \ \wedge\ p@\{7..25, 33..45\} \ \wedge\ p.dirty[h] \Rightarrow$

$$(\exists m : 0 \leq m < p.dcnt :: p.ptrs.blks[h] = p.copy[m]) \qquad \text{(I42)}$$

Proof: Initially, $p@0$ holds, so (I42) holds. Only statement $p.32$ establishes $p@\{7..25, 33..45\}$. After the execution of statement $p.32$, the antecedent does not hold because $\neg p.dirty[h]$ holds. Only statement $p.17$ establishes $p.dirty[h]$, and if it does so, the consequent holds afterwards.

No statement modifies $p.copy$ while the antecedent holds. Only statement $p.17$ modifies $p.dcnt$ or $p.ptrs.blks$ while $p@\{7..25, 33..45\}$ holds. The consequent cannot be falsified by increasing $p.dcnt$. Also, statement $p.17$ modifies $p.ptrs.blks[h]$ only if executed when $p@17 \ \wedge\ p.blkidx = h$. Thus, by (I10), statement $p.17$ does not falsify the consequent while the antecedent holds. □

invariant $q@\{2..6, 32..40\} \ \wedge\ still\_valid(q) \Rightarrow q.auxcopy = AuxObj$ (I43)

Proof: Initially, $q@0$ holds, so (I43) holds. By Axiom 1, only statement $q.1$ establishes the antecedent, and the consequent holds after the execution of statement $q.1$. No statement modifies $q.auxcopy$ while the antecedent holds, and, by Axioms 2 and 3, the antecedent does not hold after the execution of any statement that modifies $AuxObj$. □

invariant $q@2 \ \wedge \ still\_valid(q) \ \Rightarrow$

$$(\forall n : 0 \leq n < q.i :: q.ptrs.blks[n] = BUF[X.pid, X.tag].blks[n]) \qquad \text{(I44)}$$

Proof: Initially, $q@0$ holds, so (I44) holds. By Axiom 1, only statement $q.1$ establishes the antecedent, and the consequent holds after the execution of statement $q.1$ because $q.i = 0$ holds.

By Claim 2, no statement falsifies the consequent by modifying BUF. By Axioms 2 and 3, the antecedent does not hold after the execution of any statement that modifies X. Only statement $q.2$ modifies $q.i$ or $q.ptrs.blks$ while the antecedent holds. By (I26), statement $q.2$ preserves the consequent if the antecedent holds. □

invariant $q@\{3..6, 32..37\} \ \wedge \ still\_valid(q) \ \Rightarrow$

$$(\forall n : 0 \leq n < B :: q.ptrs.blks[n] = BUF[X.pid, X.tag].blks[n]) \qquad \text{(I45)}$$

Proof: Initially, $q@0$ holds, so (I45) holds. By Axiom 1, only statement $q.2$ establishes the antecedent, and it does so only if executed when $q@2 \ \wedge \ still\_valid(q) \ \wedge \ q.i = B - 1$ holds. By (I26) and (I44), the consequent holds after the execution of statement $q.2$ in this case.

By Claim 2, no statement falsifies the consequent by modifying BUF. By Axioms 2 and 3, the antecedent does not hold after the execution of any statement that modifies X. Also, no statement modifies $q.ptrs.blks$ while the antecedent holds. □

invariant $p@\{7..25, 33..45\} \ \wedge \ still\_valid(p) \ \Rightarrow$

$$(\forall h : 0 \leq h < B :: p.dirty[h] \ \vee$$

$$p.ptrs.blks[h] = BUF[X, pid, X.tag].blks[h]) \qquad \text{(I46)}$$

Proof: Initially, $p@0$ holds, so (I46) holds. Only statement $p.32$ establishes $p@\{7..25, 33..45\}$. By Axiom 1, statement $p.32$ establishes the antecedent only if executed when $p@32 \wedge still\_valid(p)$ holds. By (I45), the consequent holds before, and therefore after, the execution of $p.32$ in this case. By Axiom 1, no statement establishes $still\_valid(p)$ while $p@\{7..25, 33..45\}$ holds.

By Claim 2, no statement falsifies the consequent by modifying BUF. By Axioms 2 and 3, the antecedent does not hold after the execution of any statement that modifies X. No statement falsifies $p.dirty[h]$ while the antecedent holds, and any statement that modifies $p.ptrs.blks[h]$ for some h while the antecedent holds also establishes $p.dirty[h]$. □

invariant $0 \leq m < B \wedge p@\{7..25, 33..42\} \wedge \neg p.dirty[m] \Rightarrow$

$$\neg HAS(p.tr, 0, p.dcnt, m) \qquad \text{(I47)}$$

Proof: Initially, $p@0$ holds, so (I47) holds. Only statement $p.32$ establishes $p@\{7..25, 33..42\}$, and (I47) holds after the execution of statement $p.32$ because $p.dcnt = 0$ holds. No statement establishes $\neg p.dirty[m]$ while $p@\{7..25, 33..42\}$ holds.

Only statement $p.17$ modifies $p.tr$ or $p.dcnt$ while the antecedent holds. Statement $p.17$ falsifies the consequent only if executed when $p.blkidx = m$ holds. However, $p.17$ also falsifies the antecedent in this case. □

invariant $p@\{15..17\} \Rightarrow \neg HAS(p.tr, 0, p.dcnt, p.blkidx)$ (I48)

Proof: Initially, $p@0$ holds, so (I48) holds. Only statement $p.14$ establishes the antecedent, and, by (I47), the consequent holds after statement $p.14$ establishes the antecedent. Also, no statement falsifies the consequent while the antecedent holds. $\square$

invariant $p@\{7..25, 33..45\} \ \wedge \ still\_valid(p) \ \wedge \ 0 \leq m < p.dcnt \ \wedge \ p.oldlst[m] = n \Rightarrow$

$$BUF[X.pid, X.tag].blks[p.tr[m]] = n \ \wedge \ \neg HAS(p.copy, 0, M, n) \qquad \text{(I49)}$$

Proof: Initially, $p@0$ holds, so (I49) holds. By Axiom 1, only statement $p.32$ establishes $p@\{7..25, 33..45\} \ \wedge \ still\_valid(p)$. The antecedent does not hold after the execution of statement $p.32$ because $p.dcnt = 0$ holds. Only statement $p.17$ modifies $p.oldlst$ or $p.dcnt$ while $p@\{7..25, 33..45\}$ holds. Statement $p.17$ can establish the antecedent only if executed when $p@17 \ \wedge \ p.dcnt = m$ holds. In this case, the antecedent holds after the execution of statement $p.17$ only if $p.ptrs.blks[p.blkidx] = n$ holds before. Also, by Axiom 1, the antecedent does not hold after the execution of statement $p.17$ in this case unless $still\_valid(p)$ holds before. By (I8), (I10), and (I46), this implies $p.dcnt = m \ \wedge \ 0 \leq p.blkidx < B \ \wedge$ $BUF[X.pid, X.tag].blks[p.blkidx] = n$, which implies $HAS(BUF[X.pid, X.tag].blks, 0, B, n)$. Therefore, (I53) implies $\neg HAS(p.copy, 0, M, n)$. Thus, the first conjunct of the consequent holds after the execution of statement $p.17$, and the second conjunct holds before, and therefore after, the execution of statement $p.17$.

No statement modifies $p.copy$ while the antecedent holds. Only statement $p.17$ modifies $p.tr$ while the antecedent holds, and it modifies $p.tr[m]$ only if executed when $p@17 \ \wedge \ p.dcnt = m$ holds. As shown above, either the antecedent does not hold after the execution of statement $p.17$ in this case, or the consequent does hold. By Axioms 2 and 3, the antecedent does not hold after the execution of any statement that modifies X, and, by Claim 2, no statement falsifies the consequent by modifying BUF. $\square$

invariant $p@46 \ \wedge\ 0 \leq m < p.dcnt \ \wedge\ p.oldlst[m] = n \Rightarrow$

$$\neg HAS(p.copy, p.dcnt, M, n) \qquad \text{(I50)}$$

Proof: Initially, $p@0$ holds, so (I50) holds. No statement modifies $p.dcnt$ or $p.oldlst$ while $p@46$ holds. Only statement $p.11$ establishes $p@46$. By Axiom 2 and (I49), if statement $p.11$ establishes the antecedent then the consequent holds before the execution of $p.11$. Statement $p.11$ does not falsify the consequent. Also, no statement falsifies the consequent while the antecedent holds. □

invariant $((p@\{7..25, 33..45\} \ \wedge\ still\_valid(p)) \ \vee\ p@46) \Rightarrow UNIQ(p.oldlst, p.dcnt)$ (I51)

Proof: Initially, $p@0$ holds, so (I51) holds. By Axiom 1, no statement establishes $still\_valid(p)$ while $p@\{7..25, 33..46\}$ holds. Only statement $p.11$ establishes $p@46$, and, by Axiom 2, it does so only if $p@11 \ \wedge\ still\_valid(p)$ holds beforehand, in which case, the antecedent already holds. Only statement $p.32$ establishes $p@\{7..25, 33..45\}$. The consequent holds after the execution of statement $p.32$ because $p.dcnt = 0$ holds.

Only statement $p.17$ modifies $p.oldlst$ or $p.dcnt$ while the antecedent holds, and, by Axiom 1, it does so only if executed when $p@17 \ \wedge\ still\_valid(p)$ holds. Also, statement $p.17$ falsifies the consequent only if executed when $(\exists m : 0 \leq m < p.dcnt :: p.oldlst[m] = p.ptrs.blks[p.blkidx])$ holds. By (I10), (I46), and (I49), this implies that $(\exists m : 0 \leq m < p.dcnt :: BUF[X.pid, X.tag].blks[p.tr[m]] = BUF[X.pid, X.tag].blks[p.blkidx])$ holds. However, because (I48) implies that $p.tr[m] \neq p.blkidx$, (I8) and (I55) imply that this does not hold. Thus, statement $p.17$ does not falsify the consequent. □

invariant $UNIQ(p.copy, M)$ (I52)

Proof: Initially, (I52) holds. Only statement $p.46$ modifies $p.copy$. By (I50) and (I51), statement $p.46$ does not falsify (I52). $\square$

invariant $p \neq q \;\wedge\; ((p@46 \;\wedge\; HL(p,n)) \;\vee\; (\neg p@46 \;\wedge\; HAS(p.copy, 0, M, n))) \Rightarrow$

$$(\neg q@46 \;\vee\; \neg HL(q,n)) \;\wedge\; (q@46 \;\vee\; \neg HAS(q.copy, 0, M, n)) \;\wedge$$

$$\neg HAS(BUF[X.pid, X.tag].blks, 0, B, n) \;\wedge$$

$$(\neg still\_valid(q) \;\vee\; \neg HL(q,n) \;\vee\; \neg q@\{7..25, 33..45\}) \tag{I53}$$

Proof: Initially, $p@0 \;\wedge\; q@0$ holds. If $n < pM$ or $n \geq (p+1)M$ then the antecedent does not hold initially. If $pM \leq n < (p+1)M$, then

$$\neg HAS(q.copy, 0, M, n) \;\wedge\; \neg HAS(BUF[X.pid, X.tag].blks, 0, B, n)$$

holds initially. Therefore, (I53) holds initially.

No statement modifies $p.copy$ while $\neg p@46$ holds. Only statement $p.46$ establishes $\neg p@46$, and the antecedent holds after the execution of $p.46$ only if $p@46 \;\wedge\; HL(p,n)$ holds before. Thus, statement $p.46$ does not establish the antecedent. By Claim 5, $\neg p@46$ holds after any statement execution that establishes $HL(p,n)$. Only statement $p.11$ establishes $p@46$. By Axiom 2 and Claim 5, statement $p.11$ establishes the antecedent only if executed when $p@11 \wedge still\_valid(p) \wedge HL(p,n)$ holds. By $(\text{I53})^{p,q}_{q,p}$, this implies that the first two conjuncts of the consequent hold before the execution of statement $p.11$; statement $p.11$ does not falsify either of them. Also, (I15) and (I54) imply that $\neg HAS(BUF[p, p.side].blks, 0, B, n)$ holds before the execution of statement $p.11$ in this case. Therefore, by Axiom 3, statement $p.11$ establishes the third conjunct of the consequent in this case. Finally, by Axiom 3, statement $p.11$ establishes $\neg still\_valid(q)$, so the last conjunct of the consequent holds after the execution of $p.11$. We now consider statements that potentially falsify the consequent while the antecedent holds.

By Claim 5, only statement $q.11$ can falsify the first conjunct of the consequent, and, by Axiom 2, it does so only if executed when $q@11 \ \wedge \ \mathit{still\_valid}(q)$ holds. The last conjunct of the consequent implies that $\neg HL(q,n)$ holds before, and therefore after, the execution of statement $q.11$ in this case.

Only statement $q.46$ can falsify $q@46$ or modify $q.copy$. The first conjunct of the consequent implies that $\neg HL(q,n)$ holds before the execution of statement $q.46$, which implies that $\neg HAS(q.copy, 0, M, n)$ holds afterwards.

By Claim 2, no statement falsifies the third conjunct of the consequent by modifying BUF. By Axiom 2, X is modified only by statement $s.11$, where s is any process, and only if $s.11$ is executed when $s@11 \ \wedge \ \mathit{still\_valid}(s)$ holds. If $s = p \ \wedge \ \neg HL(p,n)$ holds before the execution of $s.11$, then by Claim 5, the antecedent does not hold after the execution of statement $s.11$. If $s = p \ \wedge \ s@11 \ \wedge \ \mathit{still\_valid}(s) \ \wedge \ HL(p,n)$ holds before the execution of $s.11$, then, as shown above, the consequent holds after the execution of statement $s.11$. If $s \neq p \ \wedge \ s@11 \ \wedge \ \mathit{still\_valid}(s)$ holds before the execution of statement $s.11$, then, by $(I53)^q_s$, $s@11 \ \wedge \ \mathit{still\_valid}(s) \ \wedge \ \neg HAS(s.copy, 0, M, n)$ holds before the execution of statement $s.11$. Therefore, by (I15), (I35), (I42), and (I46), $s.11$ does not falsify the third conjunct of the consequent.

By Axiom 1, only statement $q.1$ establishes $\mathit{still\_valid}(p)$, and $q@2$ holds after the execution of statement $q.1$. By Claim 5, only statements $q.17$, $q.32$, and $q.46$ establish $HL(q,n)$. Also, only statement $q.32$ establishes $q@\{7..25, 33..45\}$. After the execution of statement $q.46$, $\neg q@\{7..25, 33..45\}$ holds. Also, by the second conjunct of the consequent, $\neg HAS(q.copy, 0, M, n)$ holds before the execution of statement $q.32$. Thus, $\neg HL(q, n)$ holds afterwards. By Axiom 1, $still\_valid(q)$ holds after the execution of statement $q.17$ if $still\_valid(q)$ holds before. By (I8) and (I45), $q@17 \ \wedge \ still\_valid(q)$ implies $q.ptrs.blks[q.blkidx] = BUF[X.pid, X.tag].blks[q.blkidx]$. Thus, by the third conjunct of the consequent, statement $q.17$ does not establish $HL(q, n)$ in this case. $\square$

invariant $p@\{7..25, 33..45\} \ \wedge \ still\_valid(p) \ \wedge \ HL(p, n) \Rightarrow$

$$\neg HAS(p.ptrs.blks, 0, B, n) \qquad \text{(I54)}$$

Proof: Initially, $p@0$ holds, so (I54) holds. No statement modifies $p.copy$ while $p@\{7..25, 33..45\}$ holds. Only statement $p.17$ modifies $p.dcnt$ or $p.oldlst$ while $p@\{7..25, 33..45\}$ holds. $HAS(p.copy, p.dcnt, M, n)$ is not established by increasing $p.dcnt$. Statement $p.17$ establishes $HAS(p.oldlst, 0, p.dcnt, n)$ only if executed when $p.ptrs.blks[p.blkidx] = n$. Thus, by Axiom 1, $p.17$ establishes the antecedent only if executed when $p@17 \wedge still\_valid(p) \wedge p.ptrs.blks[p.blkidx] = n \wedge \neg HAS(p.copy, p.dcnt, M, n)$ holds. Therefore, by (I34) and (I56), the consequent holds after the execution of statement $p.17$.

Only statement $p.32$ establishes $p@\{7..25, 33..45\}$. By Axiom 1, $p.32$ establishes the antecedent only if executed when $p@32 \ \wedge \ still\_valid(p) \ \wedge \ HAS(p.copy, 0, M, n)$ holds. By (I35), (I45), and (I53), the consequent holds before the execution of statement $p.32$ in this case; $p.32$ does not falsify the consequent.

Only statement $p.17$ modifies $p.ptrs.blks$ while the antecedent holds, and $p.17$ falsifies the consequent only if executed when $p@17 \wedge p.copy[dcnt] = n \wedge p.ptrs.blks[p.blkidx] \neq n$ holds. However, by (I49) and (I52), the antecedent does not hold after the execution of statement $p.17$ in this case. $\square$

invariant $UNIQ(BUF[X.pid, X.tag].blks, B)$ (I55)

Proof: Initially (I55) holds. By Claim 2, (I55) is potentially falsified only by statements that modify X. The only statement that modifies X is statement $s.11$, where s is any process, and, by Axiom 2, $s.11$ modifies X only if executed when $s@11 \ \wedge \ still\_valid(s)$ holds. By (I15) and (I56), (I55) holds after the execution of statement $s.11$ in this case. $\square$

invariant $p@\{7..25, 38..45\} \ \wedge \ still\_valid(p) \Rightarrow UNIQ(p.ptrs.blks, B)$ (I56)

Proof: Initially, $p@0$ holds, so (I56) holds. By Axioms 1 and 4, only statement $p.37$ establishes the antecedent, and it does so only if executed when $p@37 \ \wedge \ still\_valid(p)$ holds. By (I45) and (I55), the consequent holds before, and therefore after, the execution of statement $p.37$ in this case.

Only statement $p.17$ modifies $p.ptrs.blks$ while the antecedent holds. By Axiom 1, $p.17$ falsifies the consequent while the antecedent holds only if executed when $p@17 \ \wedge$ $still\_valid(p)$ holds. By (I34), $HAS(p.copy, 0, M, p.copy[p.dcnt])$ holds before the execution of statement $p.17$. By (I53), this implies

$$p@17 \ \wedge \ \neg HAS(BUF[X.pid, X.tag].blks, 0, B, p.copy[p.dcnt]).$$

Thus, by (I46), $p@17 \wedge (\forall h : 0 \leq h < B :: p.dirty[h] \vee p.ptrs.blks[h] \neq p.copy[p.dcnt])$ holds. However, if $p@17 \ \wedge \ p.dirty[h] \ \wedge \ p.ptrs.blks[h] = p.copy[p.dcnt]$ holds for some h, where $0 \leq h < B$, then by (I42), $p@17 \ \wedge \ (\exists m : 0 \leq m < p.dcnt :: p.copy[p.dcnt] = p.copy[m])$. By (I34), this contradicts (I52). Therefore, statement $p.17$ does not falsify the consequent. $\square$

Claim 6: Statements other than $q.18$ no not modify $BLK[x]$ for any x such that $HAS(q.ptrs.blks, 0, B, x)$ while $\neg q@46 \ \wedge \ still\_valid(q)$ holds.

Proof: Only statements $s.16$ and $s.18$, where s is any process, modify BLK. If $s = q$, then, by (I34) and (I54), statement $s.16$ does not modify $BLK[x]$ for any x such that $HAS(q.ptrs.blks, 0, B, x)$ holds while $\neg q@46 \ \wedge \ still\_valid(q)$ holds. By (I42) and (I46), $\neg q@46 \ \wedge \ HAS(q.ptrs.blks, 0, B, x)$ implies

$$\neg q@46 \ \wedge \ (HAS(p.copy, 0, p.dcnt, x) \ \vee \ HAS(BUF[X.pid, X.tag].blks, 0, B, x)).$$

Thus, by $(\mathrm{I53})_s^p$, $s@46 \vee \neg HAS(s.copy, 0, M, x)$ holds. By (I34), this implies that statement $s.16$ does not modify $BLK[x]$.

By (I8), (I11), and (I42), $\neg s@46 \ \wedge \ HAS(s.copy, 0, s.dcnt, s.ptrs.blks[s.blkidx])$ holds before the execution of statement $s.18$. Thus, by (I53) and the above assertion, statement $s.18$ does not modify $BLK[x]$ for any x such that $HAS(q.ptrs.blks, 0, B, x)$ holds.$\Box$

invariant $q@16 \ \wedge \ still\_valid(q) \Rightarrow q.tmpword = BLK[q.ptrs.blks[q.blkidx]][q.k]$ (I57)

Proof: Initially, $q@0$ holds, so (I57) holds. By Axiom 1, only statement $q.15$ establishes the antecedent; $q.15$ establishes the consequent. No statement modifies $q.tmpword$, $q.ptrs.blks$, $q.blkidx$, or $q.k$ while the antecedent holds. By (I8) and Claim 6, no statement falsifies the consequent by modifying BLK while the antecedent holds. $\Box$

invariant $s@18 \Rightarrow HAS(s.copy, 0, s.dcnt, s.ptrs.blkidx[s.blkidx])$ (I58)

Proof: Initially, $s@0$ holds, so (I58) holds. Only statements $s.14$ and $s.17$ establish the antecedent. It is easy to see that the consequent holds after the execution of statement

$s.17$. By (I7) and (I42), the consequent holds before and after statement $s.14$ establishes the antecedent. No statement falsifies the consequent while the antecedent holds. $\square$

Claim 7: Statements of processes other than q no not modify $BLK[x]$ for any x such that $HAS(q.copy, 0, M, x)$ while $\neg q@46$ holds.

Proof: Only statements $s.16$ and $s.18$, where $s \neq q$, potentially violate the claim. By (I11), (I34) and (I58), $HAS(s.copy, 0, M, x)$ holds before either statement modifies $BLK[x]$. By (I53), this implies that this does not hold while $\neg q@46 \;\wedge\; HAS(q.copy, 0, M, x)$ holds, so the claim holds. $\square$

invariant $q@\{15..16\} \;\wedge\; still\_valid(q) \Rightarrow (\forall k : 0 \leq k < q.k ::$

$$BLK[q.copy[q.dcnt]][k] = BLK[q.ptrs.blks[q.blkidx]][k]) \qquad \text{(I59)}$$

Proof: Initially, $q@0$ holds, so (I59) holds. By Axiom 1, only statement $q.14$ establishes the antecedent. After the execution of statement $q.14$, either the antecedent does not hold, or the consequent holds vacuously because $q.k = 0$ holds.

No statement modifies $q.copy$, $q.dcnt$, $q.ptrs.blks$, or $q.blkidx$ while the antecedent holds. Only statement $q.16$ modifies $q.k$ and, by (I57) and the assumption that (I59) holds before the execution of statement $q.16$, if follows that statement $q.16$ does not falsify the consequent while the antecedent holds. Also, by (I8) and (I34) and Claims 6 and 7, no statement falsifies the consequent by modifying BLK while the antecedent holds. $\square$

invariant $q@17 \;\wedge\; still\_valid(q) \Rightarrow (\forall k : 0 \leq k < S ::$

$$BLK[q.copy[q.dcnt]][k] = BLK[q.ptrs.blks[q.blkidx]][k]) \qquad \text{(I60)}$$

Proof: Initially, $q@0$ holds, so (I60) holds. By Axiom 1, only statement $q.16$ establishes the antecedent, and it does so only if executed when $q@16 \wedge \mathit{still\_valid}(q) \wedge q.k = S-1$. By (I57) and (I59), statement $q.16$ establishes the consequent in this case.

No statement modifies $q.copy$, $q.dcnt$, $q.ptrs.blks$, or $q.blkidx$ while the antecedent holds. Also, by (I8) and (I34) and Claims 6 and 7, no statement falsifies the consequent by modifying BLK while the antecedent holds. $\square$

invariant $q@\{4..25, 32..45\} \wedge \mathit{still\_valid}(q) \Rightarrow EQ(q.ptrs.blks, q.auxcopy)$ (I61)

Proof: Initially, $q@0$ holds, so (I61) holds. By Axiom 1, only statement $q.3$ establishes the antecedent, and it does so only if executed when $q@3 \wedge \mathit{still\_valid}(q)$ holds. By (I43), (I45), (I65), the consequent holds before the execution of statement $q.3$ in this case; statement $q.3$ does not falsify the consequent.

Only statement $q.17$ modifies $q.ptrs.blks$ while the antecedent holds. By Axiom 1, the antecedent holds after the execution of statement $q.17$ only if $q@17 \wedge \mathit{still\_valid}(q)$ holds before. Thus, by (I60), $q.17$ does not falsify the consequent. Only statement $q.18$ modifies $q.auxcopy$ while the antecedent holds, and by Claim 6, statements other than $q.18$ do not modify $BLK[x]$ for any x such that $HAS(q.ptrs.blks[m], 0, B, x)$ holds. It is easy to see from (I8), (I9), and the definition of EQ that statement $q.18$ does not falsify the consequent. $\square$

invariant $q@\{24..25\} \wedge \mathit{still\_valid}(q) \Rightarrow q.auxcopy =$

$$\mathit{Apply\_All}(AuxObj, q.hlplst \cdot (q.pr, q.applyop, q.applypars, q.rv)) \qquad \text{(I62)}$$

Proof: Initially, $q@0$ holds, so (I62) holds. By Axiom 1, the antecedent is established only by the return of a call to $\mathit{Simulate\_Op}$, such that $q@23 \wedge \mathit{still\_valid}(q)$ held before the call to

Simulate_Op. By (I63), this implies that $q.auxcopy = Apply\_All(AuxObj, q.hlplst)$ held before the call to *Simulate_Op*. By Axioms 1, 2, and 3, if *AuxObj* changes during the execution of *Simulate_Op*, then the return from *Simulate_Op* does not establish the antecedent. By (I9) and (I61), each $Read(q.word)$ call from *Simulate_Op* returns $q.auxcopy[q.word]$. Also, observe that each $Write(q.word, q.val)$ call from *Simulate_Op* writes $q.val$ to $q.auxcopy[q.word]$. Thus, by the assumption that $Simulate\_Op(q.applyop, q.applypars)$ makes the same sequence of calls to *Read* and *Write* as the user-supplied operation does, and returns the same value, it follows that the consequent holds after the return from *Simulate_Op* establishes the antecedent.

No statement modifies $q.auxcopy$, $q.hlplst$, $q.pr$, $q.applyop$, $q.applypars$, or $q.rv$ while the antecedent holds. Also, by Axioms 2 and 3, any statement that modifies *Auxcopy* also falsifies the antecedent. □

invariant $q@\{7..11, 20..23, 41..46\} \wedge still\_valid(q) \Rightarrow$

$$q.auxcopy = Apply\_All(AuxObj, q.hlplst) \qquad \text{(I63)}$$

Proof: Initially, $q@0$ holds, so (I63) holds. By Axiom 1, only statements $q.25$ and $q.40$ can establish the antecedent, and they do so only if executed when $still\_valid(q)$ holds. Thus, by (I43), statement $q.40$ establishes the consequent if it establishes the antecedent because it establishes $q.hlplst = \{\}$. Also, by (I62), statement $q.25$ establishes the consequent if it establishes the antecedent.

By Axioms 2 and 3, the antecedent does not hold after the execution of any statement that modifies *AuxObj*. No statement modifies $q.hlplst$ or $q.auxcopy$ while the antecedent holds. □

invariant $q@11 \ \wedge \ still\_valid(q) \Rightarrow$

$$EQ(BUF[q, q.side].blks, Apply\_All(AuxObj, q.hlplst)) \qquad \text{(I64)}$$

Proof: (I64) follows directly from (I15), (I63), and (I61). □

invariant $EQ(BUF[X.pid, X.tag].blks, AuxObj)$ (I65)

Proof: Initially, (I65) holds. By Claim 2, no statement falsifies (I65) by modifying BUF. Only statement $q.11$, where q is any process, modifies X or $AuxObj$, and, by Axiom 2, it does so only if executed when $q@11 \ \wedge \ still\_valid(q)$ holds. By (I64), statement $q.11$ does not falsify (I65) in this case. □

invariant $q@41 \Rightarrow (\forall p : p \neq q :: \neg(\exists \ tup \in q.hlplst :: tup.pid = p))$ (I66)

Proof: Initially, $q@0$ holds, so (I66) holds. Only statements $q.21$ and $q.25$ establish the antecedent, and they do so only if executed when $q.from = 40$ holds. By (I20), the consequent holds after the execution of either statement in this case. No statement modifies $q.hlplst$ while the antecedent holds. □

invariant $q@43 \ \wedge \ (\exists \ tup \in q.hlplst :: tup.pid = p) \Rightarrow p = q \ \vee$

$$0 \leq (p + N - q.ptrs.help) \ \mathbf{mod} \ N \leq (q.try + N - q.ptrs.help) \ \mathbf{mod} \ N \qquad \text{(I67)}$$

Proof: Initially, $q@0$ holds, so (I67) holds. Only statements $q.21$ and $q.25$ establish $q@43$, and only statements $q.25$ establishes $(\exists \ tup \in q.hlplst :: tup.pid = p)$. By (I68), if $q.21$ establishes the antecedent, then the consequent holds afterwards, and by (I18) and (I68), if statement $q.25$ establishes the antecedent, then the consequent holds afterwards. No statement falsifies the consequent while the antecedent holds. □

invariant $q@\{12..25, 42\} \;\wedge\; (\exists\ tup \in q.hlplst :: tup.pid = p) \Rightarrow p = q \;\vee$

$$0 \leq (p + N - q.ptrs.help) \ \mathbf{mod}\ N < (q.try + N - q.ptrs.help) \ \mathbf{mod}\ N \qquad \text{(I68)}$$

Proof: Initially, $q@0$ holds, so (I68) holds. The antecedent does not hold after the execution of any statement that modifies $q.hlplst$. Thus, only statements $q.41$ and $q.43$ can establish the antecedent. By (I66), if $p \neq q$, then the antecedent does not hold after the execution of statement $q.41$ (if $p = q$, then the consequent holds).

If $q.try + 1 = q.ptrs.help \;\vee\; (q.try = N - 1 \;\wedge\; q.ptrs.help = 0)$, then, by (I24) and (I25), statement $q.43$ establishes $\neg q.loop$, and therefore does not establish the antecedent. Otherwise, $q.try + 1 \neq q.ptrs.help \;\wedge\; \neg(q.try = N - 1 \;\wedge\; q.ptrs.help = 0)$. By (I24) and (I25), this implies that $q.try + N - q.ptrs.help \notin \{N - 1, 2N - 1\}$, and that $1 \leq q.try + N - q.ptrs.help < 2N$. Together, these expressions imply that $(((q.try+1) \ \mathbf{mod}\ N) + N - q.ptrs.help) \ \mathbf{mod}\ N = ((q.try + N - q.ptrs.help) \ \mathbf{mod}\ N) + 1$. Therefore, if the antecedent holds after the execution of statement $q.43$, then statement $q.43$ increases $(q.try + N - q.ptrs.help) \ \mathbf{mod}\ N$ by one. Thus, by (I67), if the antecedent holds after the execution of statement $q.43$, then the consequent also holds.

No statement falsifies the consequent while the antecedent holds. □

invariant $q@\{12..25\} \Rightarrow \neg(\exists\ tup \in q.hlplst :: tup.pid = q.pr)$ (I69)

Proof: Initially, $q@0$ holds, so (I69) holds. Only statements $q.40$ and $q.42$ establish the antecedent. The consequent holds after the execution of statement $q.40$, and, by $\text{(I68)}^{p}_{q.try}$, it also holds after the execution of statement $q.42$. No statement falsifies the consequent while the antecedent holds. □

Claim 8: Only statement $p.28$ falsifies $NORM(p)$, and if it does so, then it also establishes $ST(p)$.

Proof: Statements other than $p.28$ do not modify $ANC[p]$. By Claims 2 and 3, only statement $p.49$ modifies $RET[BUF[r,t].ret][p]$ while $X = (r,t)$ holds. By (I16), statement $p.49$ does not falsify $NORM(p)$. Thus, it remains to consider statement $q.11$, where q is any process, because $q.11$ is the only statement that modifies X. By Axiom 2, $q.11$ modifies X only if executed when $q@11 \ \wedge \ still\_valid(q)$ holds. By (I31) and (I77), this implies $CPD(q, BUF[q, q.side].ret, p) \ \vee \ IBS(q, BUF[q, q.side].ret, p)$ (because $(\exists \ tup :: APD(BUF[q, q.side].ret, p, tup.val))$ implies $ST(p)$). This implies that, if $NORM(p)$ holds, then the execution of statement $q.11$ (which establishes $X = (q, q.side)$) does not falsify $NORM(p)$. It is easy to see that if $p.28$ falsifies $NORM(p)$, then it also establishes $ST(p)$.$\Box$

Claim 9: Only statement $q.11$, where q is any process, falsifies $ST(p)$, and if it does so, then it also establishes $APP(p)$ and falsifies $still\_valid(r)$ for all r.

Proof: By (I32), (I95), and the program text, no statement modifies $ANC[p]$ while $ST(p)$ holds. By Claims 2 and 3, only statement $p.49$ modifies $RET[BUF[r,t].ret][p]$ while $X = (r,t)$ holds. By (I91), statement $p.49$ does not falsify $ST(p)$. Thus, $ST(p)$ can be falsified only by modifying X. Only statement $q.11$, where q is any process, modifies X, and, by Axiom 2, it does so only if executed when $q@11 \wedge still\_valid(q)$ holds. By Axiom 3, this implies that the execution of statement $q.11$ falsifies $still\_valid(r)$ for all r in this case. It remains to show that $q.11$ also establishes $APP(p)$. By (I77), either $CPD(q, BUF[q, q.side].ret, p)$ or $APD(BUF[q, q.side].ret, p, v)$ holds for some v before the execution of statement $q.11$ in this case (because $IBS(q, BUF[q, q.side].ret, p)$ implies $q@25$).

If $CPD(q, BUF[q, q.side].ret, p) \wedge ST(p)$ holds before the execution of statement $q.11$, then by the definitions of CPD, ST, AV, and CV, it follows that statement $q.11$ does not falsify $ST(p)$. Similarly, if $APD(BUF[q, q.side].ret, p, v)$ holds for some v before the execution of statement $q.11$, then $q.11$ establishes $APP(p)$. □

Claim 10: Only statement $q.11$, where q is any process, or statement $p.49$ falsifies $APP(p)$, and if they do so, then they establish $NORM(p)$.

Proof: By (I32), (I95), and the program text, no statement modifies $ANC[p]$ while $APP(p)$ holds. By Claims 2 and 3, only statement $p.49$ modifies $RET[BUF[r,t].ret][p]$ while $X = (r,t)$ holds, and, by (I16), if it does so while $APP(p)$ holds, then it establishes $NORM(p)$. Also, only statement $q.11$, where q is any process, modifies X. Similarly to the proof of Claim 9, (I77) implies that, if statement $q.11$ falsifies $APP(p)$, then it also establishes $NORM(p)$. □

invariant $NORM(p) \vee ST(p) \vee APP(p)$ (I70)

Proof: Initially, $NORM(p)$ holds. By Claims 8, 9, and 10, no statement falsifies (I70). □

invariant $((q@21 \wedge q.match \neq RET[q.rb][q.pr].applied) \vee q@\{12..19, 22..25\}) \wedge$
$$still\_valid(q) \wedge q.pr = p \Rightarrow q.match = ANC[q.pr].bit \wedge ST(p) \qquad \text{(I71)}$$

Proof: Initially, $q@0$ holds, so (I71) holds. No statement modifies $q.match$, $q.rb$, or $q.pr$ while $q@\{12..19, 21..25\}$ holds. Only statement $q.21$ establishes $q@\{12..19, 22..25\}$, and it does so only if executed when $q@21 \wedge q.match \neq RET[q.rb][q.pr].applied$ holds. By (I41), no statement modifies $RET[q.rb]$ while $q@\{12..19, 22..25\}$ holds. Thus, by Axiom 1, only statement $q.20$ establishes the antecedent. Statement $q.20$ establishes the first conjunct of the consequent. Also, statement $q.20$ establishes the antecedent only if executed when $q@20 \ \wedge\ still\_valid(q) \ \wedge\ q.pr = p \ \wedge\ ANC[q.pr].bit \neq RET[q.rb][q.pr].applied$ holds. By (I69), (I78), (I70), and the definitions of $NORM(p)$, $ST(p)$, and $APP(p)$, this implies that $ST(p)$ holds before, and therefore after, the execution of statement $q.20$ in this case.

No statement modifies $q.match$ or $q.pr$ while the antecedent holds. Only statement $p.28$ modifies $ANC[q.pr]$ while the antecedent holds, and, by (I31) and (I95), it does not do so while the consequent holds. Also, by Claim 9, any statement that falsifies $ST(p)$ also falsifies the antecedent. $\square$

invariant $q@\{12..19, 23..25\} \ \wedge\ still\_valid(q) \ \wedge\ q.pr = p \Rightarrow q.applyop = ANC[p].op$ (I72)

Proof: Initially, $q@0$ holds, so (I72) holds. No statement modifies $q.pr$ while $q@\{12..19, 23..25\}$ holds. Therefore, by Axiom 1, only statement $q.22$ establishes the antecedent, and it does so only if executed when $q@22 \ \wedge\ still\_valid(q) \ \wedge\ q.pr = p$ holds. Statement $q.22$ establishes the consequent in this case.

No statement modifies $q.applyop$ while the antecedent holds. Also, by (I31), (I71), and (I95), no statement modifies $ANC[p]$ while the antecedent holds. $\square$

invariant $q@\{12..19, 24..25\} \ \wedge\ still\_valid(q) \ \wedge\ q.pr = p \Rightarrow$

$$q.applypars = ANC[p].pars \qquad \text{(I73)}$$

Proof: Initially, $q@0$ holds, so (I72) holds. No statement modifies $q.pr$ while $q@\{12..19, 24..25\}$ holds. Therefore, by Axiom 1, only statement $q.23$ establishes the antecedent, and it does so only if executed when $q@23 \ \wedge\ still\_valid(q) \ \wedge\ q.pr = p$ holds. Statement $q.23$ establishes the consequent in this case.

No statement modifies $q.applypars$ while the antecedent holds. Also, by (I31), (I71), and (I95), no statement modifies $ANC[p]$ while the antecedent holds. $\square$

invariant $q@\{7..25, 41..45\} \ \wedge \ still\_valid(q) \Rightarrow (\forall \ tup \in q.hlplst :: ST(tup.pid) \ \wedge$

$$tup.op = ANC[tup.pid].op \ \wedge \ tup.pars = ANC[tup.pid].pars) \qquad \text{(I74)}$$

Proof: Initially, $q@0$ holds, so (I74) holds. By Axiom 1, only statement $q.40$ establishes the antecedent; the consequent holds vacuously after the execution of statement $q.40$.

No statement modifies any tuple in $q.hlplst$ and only statement $q.25$ adds new tuples to $q.hlplst$. By Axiom 1, the antecedent holds after the execution of statement $q.25$ only if $q@25 \ \wedge \ still\_valid(q)$ holds before. By (I71), (I72), and (I73), statement $q.25$ does not falsify the consequent in this case. By Claim 9, any statement that falsifies $ST(p)$ for any p also falsifies the antecedent. Also, only statement $p.28$, where $p = tup.pid$ modifies $ANC[tup.pid]$ for any tup. However, by (I31) and (I95), this does not occur while $ST(tup.pid)$ holds. $\square$

Claim 11: No statement falsifies $CPD(q, q.rb, p) \vee IBS(q, q.rb, p)$ while $q@40 \wedge still\_valid(q)$ holds.

Proof: No statement modifies $q.rb$ while $q@40$ holds. Statements of process q are not executed while $q@40$ holds. Statements of process $s \neq q$ do not modify $q.pr$. Also, by Claim 4 and Axioms 2 and 3, any statement that modifies $RV(p)$ or $AV(p)$ also falsifies $still\_valid(q)$. For any process $s \neq q$, $(I41)^{p,q}_{q,s}$ and the program text imply that statements other than $s.49$ do not modify $RET[n]$, where $n = q.rb$ while $\neg q@46$ holds. By (I94), if statement $s.49$ modifies $RET[n]$, where $n = q.rb$, then $DND(q, s)$ holds. Because $\neg q@46$, this implies that $RET[q.rb][s].applied = ANC[s].bit$ holds. By (I16), this implies that statement $s.49$ does not falsify $CPD(q, q.rb, p) \lor IBS(q, q.rb, p)$. □

invariant $q@40 \land still\_valid(q) \Rightarrow (\forall p :: CPD(q, q.rb, p))$ (I75)

Proof: Initially, $q@0$ holds, so (I75) holds. By Axiom 1, only statement $q.39$ establishes the antecedent, and it does so only if executed when $q@39 \land still\_valid(q) \land q.j = N - 1$ holds. By (I84), (I85), and (I86), the consequent holds after the execution of statement $q.39$ in this case. Also, by Claim 11, no statement falsifies the consequent while the antecedent holds. □

Claim 12: No statement falsifies $CRV(q, q.rb, p)$ while $\neg q@46 \land still\_valid(q)$ holds.

Proof: No statement modifies $q.rb$ while $\neg q@46$ holds. Only statements $q.25$ and $q.40$ modify $q.hlplst$. By (I75), if $still\_valid(q)$ holds, then $CRV(q, q.rb, p)$ holds after the execution of statement $q.40$. Statement $q.25$ does not remove tuples from $q.hlplst$ or modify tuples already in $q.hlplst$. Thus, $q.25$ potentially violates the claim only by falsifying $\neg(\exists\ tup :: tup \in q.hlplst \land tup.pid = p) \land IBS(q, q.rb, p)$ while $still\_valid(q)$ holds (note that $CPD(q, q.rb, p)$ implies $\neg q@25 \lor q.pr \neq p$); it does so only if executed when $q@25 \land still\_valid(q) \land q.pr = p \land IBS(q, q.rb, p)$ holds. By (I71), $ST(p)$ holds before the execution of statement $q.25$ in this case. Thus, by the definitions of ST and IBS, it follows that $ANC[p].bit = (RET[q.rb][p].copied + 1)$ **mod** 3 holds before the execution of statement $q.25$. Also, by (I71), statement $q.25$ establishes $RET[q.rb][p].applied = ANC[p].bit$ in this case. Therefore, by (I71), (I72), and (I73), and Claim 9, statement $q.25$ establishes

$$(\exists\ tup :: tup \in q.hlplst \land tup.pid = p \land tup.op = ANC[p].op \land$$

$$tup.pars = ANC[p].pars \ \wedge \ APD(q.rb, p, tup.val)),$$

and therefore does not falsify $CRV(q, q.rb, p)$.

By (I31) and (I95), no statement modifies $ANC[p]$ while $ST(p)$ holds, and, by (I41), statements other than $p.49$ do not modify $RET[q.rb][p]$ while $\neg q@46$ holds. Also, by (I31), (I33), and (I97), statement $p.49$ is not executed while $ST(p)$ holds. Finally, by Claim 9, no statement falsifies $ST(p)$ while $still\_valid(q)$ holds. $\Box$

invariant $q@\{7..25, 41..45\} \ \wedge \ still\_valid(q) \Rightarrow ACRV(q, q.rb)$ (I76)

Proof: Initially, $q@0$ holds, so (I76) holds. By Axiom 1, only statement $q.40$ establishes the antecedent, and it does so only if executed when $q@40 \ \wedge \ still\_valid(q)$ holds. By (I75), statement $q.40$ establishes the consequent in this case. By Claim 12, no statement falsifies the consequent while the antecedent holds. $\Box$

invariant $q@11 \ \wedge \ still\_valid(q) \Rightarrow ACRV(q, BUF[q, q.side].ret)$ (I77)

Proof: (I77) follows directly from (I5), (I6), and (I76). $\Box$

invariant $q@\{7..25, 41..45\} \ \wedge \ still\_valid(q) \Rightarrow$

$$(\forall p :: (\exists \ tup \in q.hlplst :: tup.pid = p) \ \vee \ RET[q.rb][p].applied = AV(p)) \qquad \text{(I78)}$$

Proof: Initially, $q@0$ holds, so (I78) holds. By Axiom 1, only statement $q.40$ establishes the antecedent, and it does so only if executed when $q@40 \ \wedge \ still\_valid(q)$ holds. By (I84), the consequent holds beforehand after the execution of statement $q.40$ in this case.

Tuples are not removed from $q.hlplst$ while the antecedent holds, nor are the tuples in $q.hlplst$ modified. Also, no statement modifies $q.rb$ while the antecedent holds. By (I41), no statement modifies $RET[n][p].applied$ for any p, where $n = q.rb$, while the antecedent holds. Also, by Claim 4, $AV(p)$ for any p is falsified only if X is modified. In this case, by Axioms 2 and 3, the antecedent is also falsified. $\square$

invariant $(q@\{7..11, 41..45\} \ \vee\ (q@\{12..25\} \ \wedge\ q.pr \neq q)) \ \wedge\ still\_valid(q) \Rightarrow$

$$(APP(q) \ \vee\ NORM(q) \ \vee\ (\exists\ tup \in q.hlplst :: tup.pid = q)) \qquad \text{(I79)}$$

Proof: Initially, $q@0$ holds, so (I79) holds. By Axiom 1, no statement establishes $still\_valid(q)$ while $q@\{7..25, 41..45\}$ holds. Also, no statement modifies $q.pr$ while $q@\{12..25\}$ holds. Only statement $q.42$ establishes $q@\{12..25\} \ \wedge\ q.pr \neq q$, and statement $q.42$ does not establish the antecedent. Only statements $q.21$ and $q.25$ establish $q@\{7..11, 41..45\}$, and they establish the antecedent only if executed when $q.pr = q \ \wedge\ still\_valid(q)$ holds. It is easy to see that the consequent holds after the execution of statement $q.25$ in this case. Statement $q.21$ establishes the antecedent only if executed when $q@21 \ \wedge\ q.pr = q \ \wedge\ still\_valid(q) \ \wedge\ RET[q.rb][q].applied = q.match$ holds. By (I19) and (I78), this implies $q@21 \ \wedge\ q.pr = q \ \wedge\ q.match = AV(p)$. By (I17), this implies that $AV(q) = ANC[q].bit$, which in turn implies $\neg ST(q)$. By (I70), this implies the consequent. Statement $q.21$ does not falsify the consequent.

Tuples are not removed from $q.hlplst$ while the antecedent holds, nor are the tuples in $q.hlplst$ modified. Also, by Claims 8 and 10, $APP(q) \ \vee\ NORM(q)$ is not falsified while the antecedent holds. $\square$

invariant $q@\{33..36\} \ \wedge\ still\_valid(q) \Rightarrow (\forall r : 0 \leq r < q.h :: RET[q.rb][r].val = RV(r) \ \wedge$

$$RET[q.rb][r].applied = AV(r)) \qquad \text{(I80)}$$

Proof: Initially, $q@0$ holds, so (I80) holds. By Axiom 1, only statement $q.32$ establishes the antecedent, and the consequent holds vacuously afterwards.

No statement modifies $q.rb$ while the antecedent holds, and, by (I41), statements of process $s \neq q$ do not modify $RET[q.rb][r].applied$ or $RET[q.rb][r].val$ for any r while the antecedent holds. Also, statements of process q do not modify $RET[q.rb][r]$ for $r < q.h$ while the antecedent holds. By Claim 4 and Axioms 2 and 3, any statement that changes $RV(p)$ or $AV(p)$ also falsifies $still\_valid(q)$, thereby falsifying the antecedent. It remains to consider statements that potentially falsify the consequent by increasing $q.h$ while the antecedent holds. The only such statement is $q.36$. However, by (I83), statement $q.36$ does not falsify the consequent if executed while the antecedent holds. $\square$

invariant $q@34 \ \wedge \ still\_valid(q) \Rightarrow q.tmpval = RV(q.h)$ (I81)

Proof: Initially, $q@0$ holds, so (I81) holds. By Axiom 1, only statement $q.33$ establishes the antecedent, and it does so only if executed when $q@33 \ \wedge \ still\_valid(q)$ holds. By (I40), statement $q.33$ establishes the consequent in this case.

No statement modifies $q.tmpval$ or $q.h$ while the antecedent holds. Also, by Claim 4 and Axioms 2 and 3, any statement that changes $RV(p)$, where $p = q.h$, also falsifies $still\_valid(q)$, thereby falsifying the antecedent. $\square$

invariant $q@35 \ \wedge \ still\_valid(q) \Rightarrow RET[q.rb][q.h].val = RV(q.h)$ (I82)

Proof: Initially, $q@0$ holds, so (I82) holds. By Axiom 1, only statement $q.34$ establishes the antecedent, and it does so only if executed when $still\_valid(q)$ holds. In this case, (I81) implies that statement $q.34$ establishes the consequent.

No statement modifies $q.rb$ or $q.h$ while the antecedent holds, and, by (I41), statements of process $s \neq q$ do not modify $RET[q.rb][q.h].val$ while the antecedent holds. Also, by Claim 4 and Axioms 2 and 3, any statement that changes $RV(p)$, where $p = q.h$, also falsifies $still\_valid(q)$, thereby falsifying the antecedent. $\square$

invariant $q@36 \ \wedge \ still\_valid(q) \Rightarrow$

$$RET[q.rb][q.h].val = RV(q.h) \ \wedge \ q.tmpbit = AV(q.h) \qquad \text{(I83)}$$

Initially, $q@0$ holds, so (I83) holds. By Axiom 1, only statement $q.35$ establishes the antecedent, and it does so only if executed when $still\_valid(q)$ holds. In this case, (I40) and (I82) imply that statement $q.35$ establishes the consequent.

No statement modifies $q.rb$, $q.h$, or $q.tmpbit$ while the antecedent holds, by (I41), statements of process $s \neq q$ do not modify $RET[q.rb][q.h].val$ while the antecedent holds. Also, by Claim 4 and Axioms 2 and 3, any statement that changes $RV(p)$ or $AV(p)$, where $p = q.h$, also falsifies $still\_valid(q)$, thereby falsifying the antecedent. $\square$

invariant $q@\{37..40\} \ \wedge \ still\_valid(q) \Rightarrow (\forall r : 0 \leq r < N :: RET[q.rb][r].val = RV(r) \ \wedge$

$$RET[q.rb][r].applied = AV(r)) \qquad \text{(I84)}$$

Proof: Initially, $q@0$ holds, so (I84) holds. By Axiom 1, only statement $q.36$ establishes the antecedent, and it does so only if executed when $still\_valid(q) \ \wedge \ q.h = N - 1$ holds. In this case, (I83) and (I84) imply that statement $q.36$ establishes the consequent.

No statement modifies $q.rb$ while the antecedent holds, and, by (I41), statements of process $s \neq q$ do not modify $RET[q.rb][r].val$ or $RET[q.rb][r].applied$ for any r while the antecedent holds. Also, statements of process q do not modify $RET[q.rb]$ while the antecedent holds. By Claim 4 and Axioms 2 and 3, any statement that changes $RV(p)$ or $AV(p)$ also falsifies $still\_valid(q)$, thereby falsifying the antecedent. $\square$

invariant $q@39 \Rightarrow q.a = RET[q.rb][q.j].applied$ (I85)

Proof: Initially, $q@0$ holds, so (I85) holds. Only statement $q.38$ establishes the antecedent, and statement $q.38$ establishes the consequent. No statement modifies $q.rb$ or $q.j$ while the antecedent holds, and, by (I41), no statement modifies $RET[n][p].applied$ where $n = q.rb \ \wedge \ r = q.j$ while the antecedent holds. $\square$

invariant $q@\{38..39\} \Rightarrow (\forall r : 0 \leq r < q.j ::$

$$RET[q.rb][r].copied = RET[q.rb][r].applied) \qquad \text{(I86)}$$

Proof: Initially, $q@0$ holds, so (I86) holds. Only statement $q.37$ establishes the antecedent, and the consequent holds vacuously after the execution of statement $q.37$. By (I41), only statements $q.39$ and $r.49$ modify $RET[q.rb][r]$ while the antecedent holds. Also, only statement $q.39$ modifies $q.j$ while the antecedent holds. By (I85), statement $q.39$ preserves the consequent. Also, by (I94), statement $r.49$ writes $RET[q.rb][r].copied$ only if $DND(q, r)$ holds. Because $q@\{38..39\}$, this implies that $RET[q.rb][r].applied = ANC[r].bit$ holds. By (I16), this implies that statement $r.49$ does not falsify the consequent in this case. $\square$

Claim 13: No statement falsifies $STS(p, n)$ while $(\forall q :: q.rb \neq n \ \vee \ WND(q, p)) \wedge ST(p) \wedge \neg p@\{28, 49\}$ holds.

Proof: No statement modifies $ANC[p]$ while $\neg p@28$ holds. Only statement $q.39$, where q is any process, and statement $p.49$ modify $RET[n][p].copied$. Statement $p.49$ is not executed when $\neg p@\{28, 49\}$ holds. Statement $q.39$ does not modify $RET[n][p].copied$ if executed when $q.j \neq p$ holds. Finally, $q@39 \wedge q.j = p \wedge WND(q,p)$ implies $(RET[q.rb][p].applied + 1) \textbf{ mod } 3 = ANC[p].bit$. Thus, by (I85), $(q.a + 1) \textbf{ mod } 3 = ANC[p].bit$ holds before the execution of statement $q.39$, so $q.39$ does not falsify $STS(p,n)$ in this case. $\square$

Claim 14: For any n, no statement falsifies $WND(q,p)$ for any $q \neq p$ while $(\forall q :: q.rb \neq n \vee WND(q,p)) \wedge ST(p) \wedge \neg p@\{28, 49\}$ holds.

Proof: Only statement $q.37$ establishes $q@\{12..25, 38..44\}$, and, by Axiom 4, it does so only if executed when $q@37 \wedge still\_valid(q)$ holds. By (I84), statement $q.37$ establishes $q@38 \wedge (RET[q.rb][p].applied+1) \textbf{ mod } 3 = ANC[p].bit$ in this case (because $ST(p)$ implies that $(AV(p) + 1) \textbf{ mod } 3 = ANC[p].bit$), and therefore does not falsify $WND(q,p)$.

Only statement $q.39$ falsifies $q@\{38..39\}$, and it does so only if executed when $q.j = N - 1$ holds. By (I85) and (I86), $q.39$ establishes $q@40 \wedge STS(p, q.rb)$ if executed when $q@39 \wedge q.j = N - 1 \wedge (RET[q.rb][p].applied + 1) \textbf{ mod } 3 = ANC[p].bit$ holds. By (I41), no statement modifies $RET[q.rb][p].applied$ while $q@\{38..39\}$ holds. Also, no statement modifies $ANC[p]$ while $\neg p@28$ holds.

Only statement $q.44$ falsifies $q@\{12..25, 40..44\}$, and $\neg q@\{12..25, 38..44\}$ holds after the execution of statement $q.44$. No statement modifies $q.rb$ while $q@\{12..25, 40..44\}$ holds, and, by Claim 13, no statement falsifies $STS(p,n)$, where $n = q.rb$, while $(\forall q :: q.rb \neq n \vee WND(q,p)) \wedge ST(p) \wedge \neg p@\{28, 49\}$ holds. $\square$

invariant $p@27 \wedge ST(p) \wedge p.from \in \{29, 47\} \wedge p.tmp = N \Rightarrow$

$$STS(p, p.ptrs.ret) \wedge (\forall q : q \neq p :: q.rb \neq p.ptrs.ret \vee WND(q,p)) \qquad \text{(I87)}$$

Proof: Initially, $p@0$ holds, so (I87) holds. No statement modifies $p.tmp$ or $p.from$ while $p@27$ holds. Claims 8 and 10 and (I70) imply that only statement $p.28$ establishes $ST(p)$.

The antecedent does not hold after the execution of statement $p.28$. By (I22), only statement $p.5$ establishes $p@27 \ \wedge \ p.tmp = N$, and by Axiom 4, it does so only if executed when $p@5 \ \wedge \ still\_valid(p)$ holds. Also, statement $p.5$ establishes the antecedent only if executed when $ST(p)$ holds. By (I39) and (I41), this implies that $STS(p, p.ptrs.ret) \ \wedge \ (\forall q : q \neq p :: q.rb \neq p.ptrs.ret \ \vee \ q@46)$ holds. Statement $p.5$ does not falsify this expression, so the consequent holds after the execution of statement $p.5$ in this case.

No statement modifies $p.ptrs.ret$ while the antecedent holds. Only statement $q.46$ modifies $q.rb$, and $WND(q, p)$ holds after the execution of statement $q.46$. Also, the antecedent implies $ST(p) \ \wedge \ WND(p, p) \ \wedge \ \neg p@\{28, 49\}$. Therefore, by Claims 13 and 14, no statement falsifies the consequent while the antecedent holds. $\square$

invariant $((p@\{2..6\} \ \wedge \ \neg still\_valid(p) \ \wedge \ X.pid = r) \ \vee \ (p@27 \ \wedge \ p.tmp = r)) \ \wedge$

$$p.from \in \{29, 47\} \ \wedge \ ST(p) \Rightarrow STS(p, LAST[r]) \ \wedge \ WND(r, p) \ \wedge$$

$$(\forall q : q \neq p \ \wedge \ q \neq r :: q.rb \neq LAST[r] \ \vee \ WND(q, p)) \qquad \text{(I88)}$$

Proof: Initially, $p@0$ holds, so (I88) holds. No statement modifies $p.from$ while $p@\{2..6, 27\}$ holds. No statement modifies $p.tmp$ while $p@27$ holds, so $p@27 \ \wedge \ p.tmp = r$ is established only by statement $p.6$ when $X.pid = r$ holds, in which case, by (I28), the antecedent already holds. Also, $p@\{2..6\}$ is established only by statement $p.1$. By Axiom 1, the antecedent does not hold after the execution of statement $p.1$. Thus, the antecedent is only established by statements that establish $ST(p)$ or $\neg still\_valid(p)$ or that modify X. By Claims 8 and 10 and (I70), the antecedent does not hold after $ST(p)$ is established. By Axioms 2 and 3, it remains to consider the execution of statement $r.11$ when $still\_valid(r)$ holds. By (I13), either the first conjunct of the consequent holds after the execution of $r.11$, or $\neg ST(p)$

holds, in which case the antecedent does not hold. Also, $r@46$ holds after the execution of $r.11$, and, by (I12) and (I41), $(\forall q : q \neq p \ \wedge \ q \neq r :: q.rb \neq LAST[r] \ \vee \ q@46)$ holds before $r.11$ is executed in this case. Statement $r.11$ does not falsify this assertion. Therefore, the consequent holds after the execution of $r.11$ (because $q@46$ implies $WND(q,p)$). We now consider statements that potentially falsify the consequent while the antecedent holds.

Only statement $r.44$ modifies $LAST[r]$. By the second conjunct of the consequent, and by the definition of $WND(r,p)$, $r.44$ does not falsify the first conjunct of the consequent. Also, the second conjunct of the consequent holds after the execution of statement $r.44$, and, by (I41), $(\forall q : q \neq p \ \wedge \ q \neq r :: q.rb \neq LAST[r] \ \vee \ q@46)$ (which implies the last conjunct) holds after the execution of statement $r.44$. Only statement $q.46$ modifies $q.rb$ for some process q, and $WND(q,p)$ holds after the execution of statement $q.46$. Also, statement $q.46$ does not falsify $STS(p, LAST[r])$ or $WND(s,p)$ for any s, nor does it modify $s.rb$ for any s. Finally, the antecedent implies $WND(p,p) \wedge ST(p) \wedge \neg p@\{28,49\} \wedge (r \neq p \vee WND(r,p)$. Therefore, by Claims 13 and 14, no statement falsifies the consequent while the antecedent holds. □

invariant $p@30 \ \wedge \ ST(p) \Rightarrow STS(p, p.b) \ \wedge \ (\forall q : q \neq p :: q.rb \neq p.b \ \vee \ WND(q,p))$ (I89)

Proof: Initially, $p@0$ holds, so (I89) holds. Only statement $p.27$ establishes $p@30$, and it does so only if executed when $p@27 \ \wedge \ p.from \in \{29, 47\}$ holds. By (I29), $0 \leq p.tmp \leq N$ holds in this case. If $0 \leq p.tmp < N$, then, by (I88), statement $p.27$ establishes the consequent. If $p.tmp = N$, then, by (I87), statement $p.27$ establishes the consequent. We now consider statements that potentially falsify the consequent while the antecedent holds.

No statement modifies $p.b$ while $p@30$ holds. Only statement $q.46$ modifies $q.rb$, and $WND(q,p)$ holds after the execution of statement $q.46$. Also, the antecedent implies $WND(p,p) \wedge ST(p) \wedge \neg p@\{28,49\}$. Therefore, by Claims 13 and 14, no statement falsifies the consequent while the antecedent holds. □

invariant $p@46 \;\vee\; (p.done \;\wedge\; (p@\{30,47\} \;\vee\; (p@\{1..6,26..27\} \;\wedge\; p.from = 47))) \Rightarrow$

$$APP(p) \;\vee\; NORM(p) \qquad \text{(I90)}$$

Proof: Initially, $p@0$ holds, so (I90) holds. Only statement $p.46$ establishes $p.done$ or $p@47$, and $p.46$ does not establish the antecedent. Only statement $q.27$ establishes $q@30$, and it does so only if executed when $q.from \in \{29,47\}$ holds. Therefore, by (I30), statement $q.27$ does not establish the antecedent. No statement modifies $p.from$ while $p@\{1..6,26..27\}$ holds. Therefore, only statement $p.47$ establishes $p@\{1..6,26..27\} \wedge p.from = 47$. However, statement $p.47$ does not establish $p.done$, so it does not establish the antecedent. Thus, the antecedent is established only by establishing $p@46$. Only statement $p.11$ establishes $p@46$, and, by Axiom 2, it does so only if executed when $p@11 \;\wedge\; still\_valid(p)$ holds. By Claims 8 and 10, only statement $p.28$ falsifies the consequent. Thus, if the consequent holds before the execution of statement $p.11$, then it also holds afterwards. If the consequent does not hold before the execution of statement $p.11$, then by (I70), (I77), and (I79), $(\exists\; tup \in p.hlplst :: APD(BUF[q, q.side].ret, p, tup.val))$ holds. This implies that $APP(p)$ holds after the execution of statement $p.11$ in this case.

By Claims 8 and 10, no statement falsifies the consequent while the antecedent holds. □

invariant $p@\{48..50\} \;\vee\; (p@\{1..6,26..27\} \;\wedge\; p.from = 48) \Rightarrow$

$$APP(p) \ \vee \ NORM(p) \qquad \text{(I91)}$$

Proof: Initially, $p@0$ holds, so (I91) holds. No statement modifies $p.from$ while $p@\{1..6, 26..27\}$ holds. Therefore, only statement $p.48$ establishes $p@\{1..6, 26..27\} \ \wedge \ p.from = 48$. However, statement $p.48$ does not establish the antecedent. Only statement $p.30$ establishes $p@\{48..50\}$, and it does so only if executed when $p@30 \ \wedge \ (RET[p.b][p].copied = p.bit \ \vee \ p.done)$ holds. By (I90), $p@30 \ \wedge \ p.done$ implies the consequent. By (I16), (I70), and (I89), $p@30 \ \wedge \ RET[p.b][p].copied = p.bit$ also implies the consequent. By Claims 8 and 10, statement $p.30$ does not falsify the consequent, and no statement falsifies the consequent while the antecedent holds. □

Claim 15: No statement falsifies $SRV(p, n)$ while $(\forall q :: q.rb \neq n \ \vee \ DND(q, p)) \ \wedge \ (APP(p) \ \vee \ NORM(p))$ holds.

Proof: Only statements $s.24$ and $s.34$, where s is any process, potentially modify $RET[n][n].val$. However, they do not do so while $s.rb \neq n$. Also, by Claim 4, Axiom 2, (I32), and (I77), $RV(p)$ does not change while $APP(p) \ \vee \ NORM(p)$ holds (because $CPD(q, BUF[q, q.side].ret, p)$ implies $RET[BUF[q, q.side].ret][p].val = RV(p)$). □

Claim 16: For any n, no statement falsifies $DND(q, p)$ for any $q \neq p$ while $(\forall q :: q.rb \neq n \ \vee \ DND(q, p)) \ \wedge \ (APP(p) \ \vee \ NORM(p)) \ \wedge \ \neg p@28$ holds.

Proof: Only statement $q.37$ establishes $q@\{12..25, 38..44\}$, and, by Axiom 4, it does so only if executed when $q@37 \ \wedge \ still\_valid(q)$ holds. By (I84), statement $q.37$ establishes $q@38 \wedge SRV(p, q.rb) \wedge RET[q.rb][p].applied = ANC[p].bit$ in this case (because $APP(p) \vee NORM(p)$ implies that $AV(p) = ANC[p].bit$), and therefore does not falsify $DND(q, p)$.

Only statement $q.46$ modifies $q.rb$, and $\neg q@\{12..25, 38..44\}$ holds after the execution of statement $q.46$. By Claim 15, no statement falsifies $SRV(p, n)$, where $n = q.rb$, while $(\forall q :: q.rb \neq n \ \vee \ DND(q,p)) \ \wedge \ (APP(p) \ \vee \ NORM(p))$ holds.

By (I41), no statement modifies $RET[q.rb][p].applied$ while $q@\{38..44, 12..25\}$ holds. Also, no statement modifies $ANC[p].bit$ while $\neg p@28$ holds.

Only statements $q.40$, $q.42$, and $q.44$ falsify $q@\{38..44\}$. Statements $q.40$ and $q.42$ both establish $q@20$ (which implies either $q@\{12..25\} \ \wedge \ q.pr \neq p$ or $q@20 \ \wedge \ q.pr = p$), and statement $q.44$ establishes $q@45$. Thus, none of these statements falsify $DND(q,p)$.

No statement modifies $q.pr$ while $q@\{12..25\}$ holds. Only statements $q.13$, $q.21$, and $q.25$ falsify $q@\{12..25\}$, and all of these statements establish $q@\{38..44\}$. Statement $q.20$ establishes $q@21 \wedge q.pr = p \wedge q.match = ANC[p].bit$ if executed when $q@20 \wedge q.pr = p$ holds.

No statement modifies $ANC[p]$ while $\neg p@28$ holds, and no statement modifies $q.pr$ or $q.match$ while $q@21$ holds. Finally, statement $q.21$ establishes $q@\{35, 41\}$ if executed when $q@21 \ \wedge \ q.pr = p \ \wedge \ q.match = ANC[p].bit \ \wedge \ RET[q.rb][p].applied = ANC[p].bit$ holds. □

invariant $p@27 \ \wedge \ p.from = 48 \ \wedge \ p.tmp = N \Rightarrow SRV(p, p.ptrs.ret) \ \wedge$

$$(\forall q : q \neq p :: q.rb \neq p.ptrs.ret \ \vee \ DND(q,p)) \qquad \text{(I92)}$$

Proof: Initially, $p@0$ holds, so (I92) holds. No statement modifies $p.from$ or $p.tmp$ while $p@27$ holds. By (I22), only statement $p.5$ establishes $p@27 \ \wedge \ p.tmp = N$, and, by Axiom 4, it does so only if executed when $p@5 \ \wedge \ still\_valid(p)$ holds. By (I39) and (I41), this implies that $SRV(p, p.ptrs.ret) \ \wedge \ (\forall q : q \neq p :: q.rb \neq p.ptrs.ret \ \vee \ q@46)$ holds. Statement

$p.5$ does not falsify this expression. Therefore, the consequent holds after the execution of statement $p.5$. We now consider statements that potentially falsify the consequent while the antecedent holds.

No statement modifies $p.ptrs.ret$ while $p@27$ holds. Only statement $q.46$ modifies $q.rb$, and $DND(q,p)$ holds after the execution of statement $q.46$. By (I91), the antecedent implies $DND(p,p) \wedge (APP(p) \vee NORM(p)) \wedge \neg p@28$. Therefore, by Claims 15 and 16, no statement falsifies the consequent while the antecedent holds. $\square$

invariant $((p@\{2..6\} \wedge \neg still\_valid(p) \wedge X.pid = r) \vee (p@27 \wedge p.tmp = r)) \wedge$

$$p.from = 48 \Rightarrow SRV(p, LAST[r]) \wedge DND(r,p) \wedge$$

$$(\forall q : q \neq p \wedge q \neq r :: q.rb \neq LAST[r] \vee DND(q,p)) \qquad \text{(I93)}$$

Proof: Initially, $p@0$ holds, so (I93) holds. No statement modifies $p.from$ while $p@\{2..6, 27\}$ holds. No statement modifies $p.tmp$ while $p@27$ holds, so $p@27 \wedge p.tmp = r$ is established only by statement $p.6$ when $X.pid = r$ holds. By (I28), the antecedent already holds in this case. Also, $p@\{2..6\}$ is established only by statement $p.1$. By Axiom 1, the antecedent does not hold after the execution of statement $p.1$. Thus, the antecedent is only established by statements that establish $\neg still\_valid(p)$ or that modify X. In both cases, by Axioms 2 and 3, we need only consider the execution of statement $r.11$ when $still\_valid(r)$ holds. By (I13), the first conjunct holds after the execution of $r.11$ in this case. Also, $r@46$ holds after the execution of $r.11$, and, by (I12) and (I41), $(\forall q : q \neq p \wedge q \neq r :: q.rb \neq LAST[r] \vee q@46)$ holds before $r.11$ is executed in this case. Therefore, the consequent holds before and after the execution of $r.11$ (because $q@46$ implies $DND(q,p)$). We now consider statements that potentially falsify the consequent while the antecedent holds.

Only statement $r.44$ modifies $LAST[r]$. By the second conjunct of the consequent, and by the definition of $DND(r,p)$, $r.44$ does not falsify the first conjunct of the consequent. Also, the second conjunct of the consequent holds after the execution of statement $r.44$, and, by (I41), $(\forall q : q \neq p \ \wedge \ q \neq r :: q.rb \neq LAST[r] \ \vee \ q@46)$ (which implies the last conjunct) holds after the execution of statement $r.44$. Only statement $q.46$ modifies $q.rb$ for some process q, and $DND(q,p)$ holds after the execution of statement $q.46$. Also, statement $q.46$ does not falsify $SRV(p, LAST[r])$ or $DND(s,p)$ for any s, nor does it modify $s.rb$ for any $s \neq q$. Finally, by (I91), the antecedent implies $DND(p,p) \wedge (APP(p) \vee NORM(p)) \wedge \neg p@28$. Therefore, by Claims 15 and 16, no statement falsifies the consequent while the antecedent holds. $\square$

invariant $p@\{49..50\} \Rightarrow SRV(p, p.b) \ \wedge \ (\forall q : q \neq p :: q.rb \neq p.b \ \vee \ DND(q,p))$ (I94)

Proof: Initially, $p@0$ holds, so (I94) holds. Only statement $p.27$ establishes $p@\{49..50\}$, and it does so only if executed when $p@27 \wedge p.from = 48$ holds. By (I29), $0 \leq p.tmp \leq N$ holds in this case. If $0 \leq p.tmp < N$, then, by (I93), statement $p.27$ establishes the consequent. If $p.tmp = N$, then, by (I92), statement $p.27$ establishes the consequent. We now consider statements that potentially falsify the consequent while the antecedent holds.

No statement modifies $p.b$ while $p@\{49..50\}$ holds. $DND(q,p)$ holds after the execution of any statement that modifies $q.rb$. Also, by (I91), the antecedent implies $DND(p,p) \ \wedge \ (APP(p) \ \vee \ NORM(p)) \ \wedge \ \neg p@28$. Therefore, by Claims 15 and 16, no statement falsifies the consequent while the antecedent holds. $\square$

invariant $p@\{0, 28, 50\} \Rightarrow NORM(p)$ (I95)

Proof: Only statement $q.49$ establishes the antecedent, and, by (I16) and (I97), the consequent holds after the execution of statement $q.49$. By Claim 8, no statement falsifies the consequent without falsifying the antecedent. $\square$

The definitions below are used in the properties that follow.

$$(p@\{7..25, 29..48\} \ \vee \ (p@\{1..6, 26..27\} \ \wedge \ p.from \neq 48)) \ \wedge \ ST(p) \tag{A1}$$

$$(p@\{7..25, 29..48\} \ \vee \ (p@\{1..6, 26..27\} \ \wedge \ p.from \neq 48)) \ \wedge \ APP(p) \tag{A2}$$

$$p@\{26, 1\} \ \wedge \ APP(p) \ \wedge \ p.from = 48 \tag{A3}$$

$$p@\{2..4\} \ \wedge \ APP(p) \ \wedge \ p.from = 48 \ \wedge \ still\_valid(p) \ \wedge \ p.curr = X \tag{A4}$$

$$p@5 \ \wedge \ APP(p) \ \wedge \ p.from = 48 \ \wedge \ still\_valid(p) \ \wedge \ p.ptrs.ret = BUF[X.pid, X.tag].ret \tag{A5}$$

$$p@27 \ \wedge \ APP(p) \ \wedge \ p.from = 48 \ \wedge \ \ \wedge \ p.ptrs.ret = BUF[X.pid, X.tag].ret \tag{A6}$$

$$p@49 \ \wedge \ APP(p) \ \wedge \ p.b = BUF[X.pid, X.tag].ret \tag{A7}$$

$$p@\{1..27, 29..49\} \ \wedge \ NORM(p) \tag{A8}$$

$\square$

$$p@28 \ \textbf{unless} \ (A1) \tag{U2}$$

Proof: Only statement $p.28$ falsifies $p@28$. By (I16) and (I95), statement $p.28$ establishes $p@29 \ \wedge \ ST(p)$, which implies (A1). $\square$

$$(A1) \ \textbf{unless} \ (A2) \tag{U3}$$

Proof: Only statements $p.11$ and $p.30$ falsify $p@\{7..25, 29..48\} \ \vee \ (p@\{1..6, 26..27\} \ \wedge \ p.from \neq 48)$. By (I16), (I89), and (I90), statement $p.30$ does not do so while $ST(p)$ holds, and by Axiom 2, statement $p.11$ does so only if executed when $p@11 \ \wedge \ still\_valid(p)$ holds.

By (I32), (I77), and (I79), if $ST(p)$ holds, then statement $p.11$ establishes $q@46 \ \wedge \ APP(p)$ in this case. Also, by Claim 9, any statement that falsifies $ST(p)$ also establishes $APP(p)$.

□

$$RV(p) = x \ \wedge \ ((\text{A2}) \vee (\text{A3}) \vee (\text{A4}) \vee (\text{A5}) \vee (\text{A6}) \vee (\text{A7}) \vee (\text{A8})) \textbf{ unless}$$
$$RV(p) = x \ \wedge \ p@50 \ \wedge \ NORM(p) \qquad \text{(U4)}$$

Proof: By Claim 4, $RV(p)$ can be modified only by modifying X. X is modified only by statement $q.11$, where q is any process, and, by Axiom 2, only if $q@11 \ \wedge \ still\_valid(q)$ holds before hand. Therefore, by (I77), $RV(p)$ does not change while the left-hand side holds. To see this, observe that (I32) implies that $APD(BUF[q, q.side].ret, p, v)$ does not hold for any v before the execution of statement $q.11$, and that $IBS(q, BUF[q, q.side].ret, p)$ does not hold while $q@11$ holds. Thus $CPD(q, BUF[q, q.side].ret, p)$ holds, so $q.11$ does not modify $RV(p)$.

If any of (A2) through (A7) is falsified as a result of $APP(p)$ being falsified, then, by Claim 10, (A8) holds afterwards. By Axioms 2 and 3, only statement $q.11$ for some process q falsifies $still\_valid(p)$ or modifies X, and it does so only if executed when $q@11 \ \wedge$ $still\_valid(q)$ holds. By (I32) and (I77), this implies that $CPD(q, BUF[q, q.side].ret, p)$ holds if this occurs when (A2) $\vee$ (A3) $\vee$ (A4) $\vee$ (A5) $\vee$ (A6) $\vee$ (A7) holds. Therefore, (A8) holds after the execution of statement $q.11$ in this case. By Claim 2, no statement falsifies (A5), (A6), or (A7) by modifying BUF. Also, $p.curr$ is not modified while (A4) holds; $p.from \neq 48$ is not falsified while (A2) holds; $p.from$ is not modified while (A3), (A4), (A5), or (A6) holds; $p.ptrs.ret$ is not modified while (A5) or (A6) holds; and $p.b$ is not modified while (A7) holds. By Claim 8, $NORM(p)$ is not falsified while (A8) holds.

If $p@\{7..25, 29..48\} \vee (p@\{1..6, 26..27\} \wedge p.from \neq 48)$ is falsified while (A2) holds, then (A3) holds afterwards. If $p@\{26, 1\}$ is falsified while (A3) holds, then, by Axiom 1, (A4) holds afterwards. If $p@\{2..4\}$ is falsified while (A4) holds, then (A5) holds afterwards. If $p@5$ is falsified while (A5) holds, then, by Axiom 4, (A6) holds afterwards. (It is easy to see that statement $p.5$ returns to statement $p.27$ if $p.from = 49$ holds.) If $p@27$ is falsified while (A6) holds, then (A7) holds afterwards. If $p@49$ is falsified while $RV(p) = x \wedge$ (A7) holds, then, by (I16), the right-hand side holds afterwards. If $p@\{1..27, 29..49\}$ is falsified while $RV(p) = x \wedge$ (A8) holds, then the right-hand side holds afterwards. $\square$

$$p@50 \ \wedge \ NORM(p) \ \wedge \ RV(p) = x \ \textbf{unless} \ p@0 \tag{U5}$$

Proof: Statement $p.50$ establishes $p@0$. By Claim 8, no statement falsifies $NORM(p)$ while $p@50$ holds, and by Claim 4, Axiom 2, and (I77), $RV(p)$ does not change while the left-hand side holds (because $CPD(q, BUF[q, q.side].ret, p)$ implies $RET[BUF[q, q.side].ret][p].val = RV(p)$). $\square$

$$\textbf{invariant} \ \neg p@\{0, 28\} \Rightarrow (p@\{1..27, 29..45, 47\} \ \wedge \ ST(p)) \vee (A2) \vee (A3) \vee (A4) \vee (A5) \vee (A6) \vee (A7) \vee (A8) \vee (p@52 \ \wedge \ NORM(p)) \tag{I96}$$

Proof: Initially, $p@0$ holds, so (I96) holds. Only statement $p.28$ establishes the antecedent, and by (U2) the consequent holds after the execution of statement $p.28$. By (U3), (U4), and (U5), no statement falsifies the consequent while the antecedent holds. $\square$

$$\textbf{invariant} \ p@49 \Rightarrow (APP(p) \ \wedge \ p.b = BUF[X.pid, X.tag].ret) \ \vee \ NORM(p) \tag{I97}$$

Proof: (I97) follows directly from (I96). $\square$

invariant $p@50 \Rightarrow RET[p.b][p].val = RV(p)$ (I98)

Proof: (I98) is implied by (I94). $\square$

Claim 17: If a statement execution linearizes an operation by process p, then $ST(p)$ holds before, and $APP(p)$ holds afterwards.

Proof: By definition, only statement $q.11$, where q is any process, linearizes operations, and, by Axiom 2, it does so only if executed when $q@11 \wedge still\_valid(q)$ holds. In this case, (I77) implies that $ST(p)$ holds before the execution of $q.11$, and $APP(p)$ holds afterwards. $\square$

Claim 18: If a statement execution falsifies $ST(p)$, then it also linearizes an operation op by process p with parameters $pars$, where $op = ANC[p].op$ and $pars = ANC[p].pars$, and afterwards, $RV(p) = x \wedge$ (A2) holds, where x is the correct return value for that operation.

Proof: By Claim 9, only statement $q.11$, where q is any process, falsifies $ST(p)$, and by Axiom 2, it does so only if executed when $q@11 \wedge still\_valid(q)$ holds. Claim 9 also implies that $APP(p)$ holds after $ST(p)$ is falsified, and, (I32), by (I91), and (I95), $(p@\{7..25, 29..48\} \vee (p@\{1..6, 26..27\} \wedge p.from \neq 48))$ holds before and after the execution of statement $q.11$ in this case. Therefore, (I77) implies that the claim holds in this case. $\square$

Linearizability: (U2), (U3), and (U4) imply that $ST(p)$ is falsified exactly once per operation invocation by process p. By Claim 17 and (I33) this implies that exactly one operation by process p is linearized per invocation by process p. By Claim 18, (U4), (U5), and (I98), the correct operation is linearized, and p returns the correct value from that operation. $\square$

Wait-Freedom: By the assumption that each sequential operation terminates, the only risk to wait-freedom is that the loop at lines 30 to 47 does not terminate for some process p that takes infinitely many steps. (The loop at lines 42 to 43 terminates after at most N iterations.) This implies that the SC operation at line 11 fails infinitely often, which, in turn, implies that there are infinitely many successful SC operation by other processes. Suppose that $ST(p)$ holds forever. Observe that each successful SC operation by some process q is guaranteed to increase (modulo N) the help counter ($BUF[X.pid, X.tag].help$) by at least one, because each sequential operation is assumed to increase $q.dcnt$ by at most T, and it is assumed that $M \geq 2 * T$. (This implies that the loop at lines 42 to 43 will be executed at least once by any operation that subsequently performs a successful SC, and therefore that operation increases the help counter.) Thus, eventually some operation, say by process r, that performs a successful SC will call $Apply(p)$. By the assumption that $ST(p)$ holds forever, and by (I69) and (I76), it follows that the check at line 21 succeeds (by (I95), $ANC[p].bit$ does not change while $ST(p)$ holds), and that process r subsequently adds a tuple for process p to $r.hlplst$. Thus, process r linearizes p's operation. By Claim 17, this contradicts the assumption that $ST(p)$ holds forever. Suppose that $APP(p)$ holds. Then by Claim 17, no operation by process p is linearized. Thus, by (I77), $NORM(p)$ holds after the execution of the next successful SC operation. It remains to consider the case in which $NORM(p)$ holds forever. Then, by (I77), for every return block v that is successfully installed as $BUF[X.tag, X.ret].ret$, $RET[v][p].copied = ANC[p].bit$ is stable. Thus, when process p reads from a return block b that has been current since $NORM(p)$ held (line 27), (I16) implies that p detects that $RET[b][p].copied = p.bit$ and therefore the loop at lines 30 to 47 terminates. $\square$

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 lines 30 to 42 is $B + N(R + C) + MS$. 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 6: 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 $\Theta(N(NR + MS + B))$ and worst-case time complexity $\Theta(\lceil N/\min(N, \lfloor M/T \rfloor)\rceil (B + N(R + C) + MS))$. $\square$

Appendix B

Correctness Proofs for Algorithms in Chapter 6

In this appendix, we provide formal correctness proofs for the two algorithms for $(k+1,k)$-exclusion presented in Section 6.4. Each algorithm is considered in a separate subsection.

B.1 Correctness Proof for Algorithm in Figure 6.8

In this section, we prove that the algorithm of Figure 6.8 is correct. As in Section 6.3, we assume the following properties, where the latter two are required to hold only if process p is nonfaulty and at most $k-1$ processes are faulty.

invariant $|\{q :: q@\{2..14\}\}| \leq k+1$ (I99)

$p@1$ *leads-to* $p@2$ (L3)

$p@15 \ \ \textit{leads-to} \ \ p@0$ (L4)

Before proving that k-Exclusion and Starvation-Freedom hold for the program of Figure 6.8, we first establish some useful properties. The first of these properties is straightforward and is stated without proof.

invariant $X = k - |\{p :: p@\{3..10\}\}|$ (I100)

We now prove that the conjunction of the following three assertions is an invariant. This, of course, implies that each of these assertions taken individually is an invariant.

$(\neg P[q] \Rightarrow Q = q) \ \wedge \ ((\exists p :: p@\{5,6,13,14\}) = (\forall i :: P[i]))$ (I101)

$|\{q :: q@\{5,6,13,14\}\}| \leq 1$ (I102)

$p@\{6,14\} \ \vee \ (p@\{7,8\} \ \wedge \ \neg P[p]) \Rightarrow Q = p$ (I103)

invariant (I101) $\wedge$ (I102) $\wedge$ (I103)

Proof: It is straightforward to check that each of (I101), (I102), and (I103) is initially true. In the remainder of the proof, we show that no statement execution falsifies any of these assertions if executed when *all* three assertions hold.

The first conjunct of (I101) could potentially be falsified by any statement that falsifies $P[q]$ or that modifies Q. The statements to check are $q.6$, $q.14$, $p.5$, and $p.13$, where p is any process. By (I103), $Q = q$ holds before (and hence after) the execution of $q.6$ or $q.14$. By the second conjunct of (I101), $P[q]$ is true before (and hence after) the execution of $p.5$ or $p.13$.

Now, consider the second conjunct of (I101). Observe that the assertion $p@\{5,6,13,14\}$ is established only by $p.4$ or $p.12$, and only if $P[p.v]$ is false. By the first conjunct of (I101), this is the only component of P that is false when $p.4$ or $p.12$ is executed. Therefore, after the execution of one of these statements, all components of P are true. $p@\{5,6,13,14\}$ is only falsified by $p.6$ or $p.14$, both of which falsify $(\forall i :: P[i])$. By (I102), both also falsify $(\exists p :: p@\{5,6,13,14\})$. Note that no statements other than $p.4$, $p.6$, $p.12$, and $p.14$ establish or falsify $(\forall i :: P[i])$, and that we have already shown that these statements do not falsify (I101).

(I102) is proved using the second conjunct of (I101). In particular, if $p@\{5,6,13,14\}$ holds for some process p, then $(\forall i :: P[i])$ holds. Hence, $q@\{5,6,13,14\}$ cannot be established for another process q.

The antecedent of (I103) is only established by $p.5$ or $p.13$, each of which also establishes the consequent. The consequent of (I103) is only falsified by $q.5$ or $q.13$, where $q \neq p$. By (I102), $q@\{5,13\}$ implies that $p@\{6,14\}$ is false. By the second conjunct of (I101), $q@\{5,13\}$ also implies that $p@\{7,8\} \wedge \neg P[p]$ is false. Thus, if either $q.5$ or $q.13$ is enabled, then the antecedent of (I103) is false. □

$$\textbf{invariant } X < 0 \Rightarrow (\exists r :: r@3 \ \vee \ (r@4 \ \wedge \ r.v = Q) \ \vee \ r@\{5,6\} \ \vee \ (r@\{7,8\} \ \wedge \ \neg P[r])) \tag{I104}$$

Proof: The antecedent of (I104) is initially false, and is established only by $p.2$, which also establishes $p@3$. In the remainder of the proof, we check those statements that may falsify a disjunct of the consequent if executed when the antecedent holds. We show that if such a disjunct is falsified, then another disjunct holds.

$p@3$ can only be falsified by $p.3$, which also establishes $p@4 \ \wedge\ p.v = Q$.

$p@4 \ \wedge\ p.v = Q$ can be falsified only by $p.4$ or $q.5$ or $q.13$, where $q \neq p$. If $p.4$ is executed when $(\forall i :: P[i])$ holds, then by the second conjunct of (I101), $(\exists q :: q\{5, 6, 13, 14\})$ holds. If $X < 0$ holds, then, by (I99) and (I100), $(\exists q :: q\{5, 6\})$ holds, which implies that $p.4$ does not falsify (I104). If $p.4$ is executed when $(\forall i :: P[i])$ does not hold, then by the first conjunct of (I101), $P[p.v]$ is false (since $p.v = Q$). Hence, $p.4$ establishes $p@5$. Finally, $q.5$ establishes $q@6$, and if $X < 0$, then (I99) and (I100) imply that $q.13$ is not enabled.

$p@\{5, 6\}$ is falsified only by $p.6$, which establishes $p@\{7, 8\} \ \wedge\ \neg P[p]$.

If $X < 0$, then $p@\{7, 8\} \ \wedge\ \neg P[p]$ can only be falsified by $q.4$ or $q.12$ for some $q \neq p$. However, if $q.4$ establishes $P[p]$, then it also establishes $q@5$. Also, because $X < 0$, (I99) and (I100) imply that $q.12$ is not enabled. $\Box$

The following invariant establishes the k-Exclusion property for the program of Figure 6.8.

invariant $|\{p :: p@9\}| \leq k$ (I105)

Proof: If $X \geq 0$, then by (I100), $|\{p :: p@\{3..10\}\}| \leq k$ holds, so (I105) holds. If $X < 0$, then by (I104), $(\exists p :: p@\{3..8\})$ holds, so by (I99), (I105) holds. $\Box$

The following unless property, which follows directly from the program text, is used in the proof of Starvation-Freedom.

$p@8 \ \wedge\ P[p]$ *unless* $p@9$ (U6)

Starvation-Freedom: If process p is nonfaulty and at most $k - 1$ processes are faulty, then $p@1$ *leads-to* $p@9$.

Proof: By (L3) and (L4), the only risk to Starvation-Freedom is that a nonfaulty process p is blocked forever at $p.8$. Process p only reaches $p.8$ by executing $p.7$ when $X < 0$ holds. By (I100), this implies that $|\{p :: p@\{3..10\}\}| > k$ holds when $p.7$ is executed. By the assumption that at most $k-1$ processes are faulty, this implies that there is a nonfaulty process $q \neq p$ such that $q@\{3..10\}$ holds when $p.7$ is executed.

We now show that $p@8 \ \wedge \ P[p]$ holds at some state after $p.7$ is executed. Assume, to the contrary, that $p@8 \ \wedge \ \neg P[p]$ holds continually after $p.7$ is executed. Note that, by the first conjunct of (I101), this implies that process q is not blocked at $q.8$ because $q \neq p$. Because $p@8 \wedge \neg P[p]$ continues to hold, by (I103), $p@8 \wedge \neg P[p] \wedge Q = p$ continues to hold. Hence, because process q is nonfaulty and does not become blocked at $q.8$, $q.11$ is eventually executed when $p@8 \wedge \neg P[p] \wedge Q = p$ holds, establishing $q@12 \wedge q.v = Q \wedge \neg P[p] \wedge Q = p$. Statement $q.12$ is then eventually executed, establishing $P[p]$. Thus, $p@8 \ \wedge \ P[p]$ holds at some state after $p.7$ is executed.

To conclude the proof, observe that once $p@8 \wedge P[p]$ holds, by (U6), $p@9$ eventually holds, because p is nonfaulty. Thus, Starvation-Freedom holds for the program of Figure 6.8. $\square$

B.2 Correctness Proof for Algorithm in Figure 6.9

In this section, we prove that the algorithm of Figure 6.9 is correct. We begin by repeating the following lemma, which was proved in Section 6.4.2.

Lemma 2: For any statement s in the noncritical or critical section, or $Acquire(N, k+1)$ or $Release(N, k+1)$ procedure of any process q, the following properties hold, where b ranges over $\{false, true\}$.

$$\{q \neq p \ \wedge \ P[p] = (b, IN)\} \ q.s \ \{P[p] = (b, IN)\} \tag{S1}$$

$$\{q \neq p \ \wedge \ R[p] = (b, IN)\} \ q.s \ \{R[p] = (b, IN)\} \tag{S2}$$

$$\{q = p \ \wedge \ P[p] = (b, IN)\} \ q.s \ \{P[p] = (b, IN) \ \vee \ P[p].instance \neq IN\} \tag{S3}$$

$$\{q = p \ \wedge \ R[p] = (b, IN)\} \ q.s \ \{R[p] = (b, IN) \ \vee \ R[p].instance \neq IN\} \tag{S4}$$

$$\{P[p].instance \neq IN\} \ q.s \ \{P[p].instance \neq IN\} \tag{S5}$$

$$\{R[p].instance \neq IN\} \ q.s \ \{R[p].instance \neq IN\} \tag{S6}$$

□

As before, we assume the following properties concerning $Acquire(N, k+1)$ and $Release(N, k+1)$, where the latter two are required to hold only if process p is nonfaulty and at most $k-1$ processes are faulty.

$$\textbf{invariant } |\{q :: q@\{2..16\}\}| \leq k+1 \tag{I106}$$

$$p@1 \ \ leads\text{-}to \ \ p@2 \tag{L5}$$

$$p@17 \ \ leads\text{-}to \ \ p@0 \tag{L6}$$

Now that we have stated the assumptions that we require of the noncritical and critical sections and $Acquire(N, k+1)$ and $Release(N, k+1)$ procedures, we proceed to prove that k-Exclusion and Starvation-Freedom hold for the program of Figure 6.9. We first prove a number of useful intermediate properties. The first three of these properties are straightforward and are stated without proof.

$$\textbf{invariant } X = k - |\{p :: p@\{3..14\}\}| \tag{I107}$$

$$\textbf{invariant } Z = (\exists q :: q@\{4..8\}) \tag{I108}$$

invariant $|\{q :: q@\{4..8\}\}| \leq 1$ (I109)

Note that (I108) and (I109) must be proved together as a conjunction.

invariant $p@\{5,6\} \Rightarrow p.v = Q$ (I110)

Proof: (I110) clearly holds initially. The antecedent of (I110) is established only by $p.4$, which also establishes the consequent. The consequent can be falsified while the antecedent holds only by $q.6$, where $q \neq p$. However, if the antecedent holds, then by (I109), statement $q.6$ is not enabled. □

invariant $p@6 \Rightarrow P[p.v] \neq (\mathit{false}, \mathit{IN})$ (I111)

Proof: (I111) clearly holds initially. The antecedent is established only by statement $p.5$. If $p.5$ is executed when $P[v] = (\mathit{false}, \mathit{IN})$ holds, then it establishes $P[v] = (\mathit{true}, \mathit{IN})$. If $p.5$ is executed when $P[v] \neq (\mathit{false}, \mathit{IN})$, then it does not modify $P[v]$. In either case, $P[v] \neq (\mathit{false}, \mathit{IN})$ holds after the execution of $p.5$.

Now, consider statements that may falsify the consequent $P[p.v] \neq (\mathit{false}, \mathit{IN})$. Note that this expression is equivalent to $(P[p.v] = (\mathit{true}, \mathit{IN})) \vee (P[p.v].\mathit{instance} \neq \mathit{IN})$. Let $r = p.v$. By (S1) and (S5), it follows that the consequent is not falsified by any statement within the noncritical or critical section or $\mathit{Acquire}(N, k+1)$ or $\mathit{Release}(N, k+1)$ procedure of any process $q \neq r$. Furthermore, by (S3), if process r modifies $P[r]$ in one of these sections or procedures when $p.v = r$ holds, then it establishes $P[p.v].\mathit{instance} \neq \mathit{IN}$. The remaining statements to consider are $p.4$ and $p.15$, which may modify $p.v$, $r.7$, which may modify $P[p.v]$ (recall that $r = p.v$), and $q.5$, where $q.v = p.v$, which may also modify $P[p.v]$. However, the antecedent of (I111) is false after the execution of $p.4$ or $p.15$. By (I109), it is also false after the execution of $r.7$ or $q.5$, where $q \neq p$. If $q = p$, then $P[v] \neq (\mathit{false}, \mathit{IN})$ holds after the execution of $q.5$, as explained in the previous paragraph. $\Box$

invariant $p@\{7, 8\} \Rightarrow Q = p$ (I112)

Proof: (I112) clearly holds initially. The antecedent is established only by statement $p.6$, which also establishes the consequent. The consequent can only be falsified by statement $q.6$, where $q \neq p$. However, by (I109), the antecedent of (I112) is false after the execution of $q.6$. $\Box$

invariant $p@8 \Rightarrow P[p] = (\mathit{false}, \mathit{IN})$ (I113)

Proof: (I113) clearly holds initially. The antecedent is established only by $p.7$, which also establishes the consequent. If the antecedent holds, then by (S1), the consequent could potentially be falsified only by $q.5$, where $q \neq p$. However, by (I109), $q.5$ is not enabled when the antecedent holds. $\Box$

invariant $p@\{9..12\} \Rightarrow ((\neg Z \ \vee \ (\exists r :: r@\{4, 5\})) \ \wedge \ Q = p) \ \vee \ P[p] \neq (\mathit{false}, \mathit{IN})$ (I114)

Proof: (I114) clearly holds initially. The antecedent of (I114) is established only by statement $p.8$. By (I112), $p.8$ also establishes $\neg Z \ \wedge \ Q = p$. In the remainder of the proof, we consider the two disjuncts of the consequent. We show that if one disjunct is falsified while the antecedent holds, then the other disjunct holds.

We first dispose of the second disjunct, i.e., $P[p] \neq (\mathit{false}, \mathit{IN})$. This expression is equivalent to $(P[p] = (\mathit{true}, \mathit{IN})) \vee (P[p].\mathit{instance} \neq \mathit{IN})$. By (S1) and (S5), this expression is not falsified by any statement within the noncritical or critical section or $\mathit{Acquire}(N, k+1)$ or $\mathit{Release}(N, k+1)$ procedure of any process $q \neq p$. If it is falsified by process p in one of these sections or procedures, then the antecedent $p@\{9..12\}$ is false. The remaining statement to consider is $p.7$. However, the antecedent of (I114) is false after the execution of $p.7$.

We now consider the first disjunct, i.e., $(\neg Z \ \vee \ (\exists r :: r@\{4,5\})) \ \wedge \ Q = p$. This expression could potentially be falsified by any statement that establishes Z or $Q \neq p$, or that falsifies $(\exists r :: r@\{4,5\})$. We consider such statements in turn.

Z is established only by statement $q.3$ for some process q. However, if this statement is executed when $\neg Z \ \wedge \ Q = p$ holds, then it establishes $(\exists r :: r@\{4,5\}) \ \wedge \ Q = p$.

$Q \neq p$ is established only by statement $q.6$, where $q \neq p$. However, by (I110) and (I111), if $q@6 \ \wedge \ Q = p$ holds, then $P[p] \neq (\mathit{false}, \mathit{IN})$ also holds, and $q.6$ does not falsify this expression.

Finally, $(\exists r :: r@\{4,5\})$ is falsified only by statement $r.5$. However, if this statement is executed when $(\exists r :: r@\{4,5\}) \wedge Q = p \wedge P[p] = (\mathit{false}, \mathit{IN})$ holds, then by (I110), it establishes $P[p] = (\mathit{true}, \mathit{IN})$. □

invariant $p@\{11,12\} \ \wedge \ P[p] = (\mathit{false}, \mathit{IN}) \Rightarrow Q = p$ (I115)

Proof: Follows directly from (I114). □

$$\textbf{invariant } X < 0 \Rightarrow (\exists r :: r@\{3..8\} \ \vee \ (r@9 \ \wedge \ P[r] = (\mathit{false}, \mathit{IN})) \ \vee$$
$$(r@\{10..12\} \ \wedge \ P[r] = (\mathit{false}, \mathit{IN}) \ \wedge \ R[r] = (\mathit{false}, \mathit{IN}))) \qquad \text{(I116)}$$

Proof: The antecedent of (I116) is initially false, and is established only by $p.2$. However, if $p.2$ establishes $X < 0$, then it also establishes $p@3$. In the remainder of the proof, we check those statements that may falsify a disjunct of the consequent. We show that if such a disjunct is falsified, then another disjunct holds or the antecedent is false.

The first disjunct, $r@\{3..8\}$, can only be falsified by statements $r.3$ and $r.8$. However, $r.3$ falsifies $r@\{3..8\}$ only if executed when $Z = true$, which by (I108), implies that $(\exists q :: q@\{4..8\})$ holds. Also, by (I113), $r.8$ establishes $r@9 \ \wedge \ P[r] = (false, IN)$.

By (S1), the second disjunct, $r@9 \ \wedge \ P[r] = (false, IN)$, can only be falsified by statements $r.9$ and $q.5$, where q is any process. (Note that process r could potentially falsify $P[r] = (false, IN)$ in its noncritical or critical sections or $Acquire(N, k+1)$ or $Release(N, k+1)$ procedures, but in this case $r@9$ is false.) However, if $r.9$ is executed when $r@9 \wedge P[r] = (false, IN)$ holds, then it establishes $r@\{10..12\} \ \wedge \ P[r] = (false, IN) \ \wedge \ R[r] = (false, IN)$. Also, $q@\{3..8\}$ holds after the execution of $q.5$.

Finally, consider the third disjunct, i.e., $r@\{10..12\} \wedge P[r] = (false, IN) \wedge R[r] = (false, IN)$. By (S1) and (S2), this disjunct can only be falsified by $r.10$, $q.5$, and $q.16$, where q is any process. (As before, process r could potentially falsify $P[r] = (false, IN)$ or $R[r] = (false, IN)$ in its noncritical or critical sections or $Acquire(N, k+1)$ or $Release(N, k+1)$ procedures, but in this case $r@\{10..12\}$ is false.) However, $r.10$ does not falsify $r@\{10..12\}$ while the antecedent holds. Also, $q@\{3..8\}$ holds after the execution of $q.5$, and by (I106) and (I107), $q.16$ is not enabled while the antecedent holds. □

The following invariant establishes the k-Exclusion property for the program of Figure 6.9.

invariant $|\{p :: p@13\}| \leq k$ (I117)

Proof: If $X \geq 0$, then by (I107), $|\{p :: p@\{3..9\}\}| \leq k$ holds, so (I117) holds. If $X < 0$, then by (I116), $(\exists p :: p@\{3..12\})$ holds, so by (I106), (I117) holds. □

The following unless property is used in the proof of Starvation-Freedom.

$$p@\{11,12\} \ \wedge\ (P[p] \neq (\mathit{false}, \mathit{IN})\ \vee\ R[p] \neq (\mathit{false}, \mathit{IN}))\ \ \mathit{unless}\ \ p@13 \tag{U7}$$

Proof: By (S1), (S2), (S5), (S6), and the program text, neither $P[p] \neq (\mathit{false}, \mathit{IN})$ nor $R[p] \neq (\mathit{false}, \mathit{IN})$ can be falsified while $p@\{11,12\}$ holds. □

Starvation-Freedom: If process p is nonfaulty and at most $k-1$ processes are faulty, then $p@1$ *leads-to* $p@13$.

Proof: By (L5) and (L6), the only risk to Starvation-Freedom is that a nonfaulty process p is blocked forever at $p.11$ and $p.12$. Process p only reaches $p.11$ by executing $p.10$ when $X < 0$ holds. By (I107), this implies that $|\{p :: p@\{3..14\}\}| > k$ holds when $p.11$ is executed. By the assumption that at most $k-1$ processes are faulty, this implies that there is a nonfaulty process $q \neq p$ such that $q@\{3..14\}$ holds when $p.10$ is executed.

We now show that $p@\{11,12\}\ \wedge\ (P[p] \neq (\mathit{false}, \mathit{IN})\ \vee\ R[p] \neq (\mathit{false}, \mathit{IN}))$ holds at some state after $p.10$ is executed. Assume, to the contrary, that $p@\{11,12\}\ \wedge\ P[p] = (\mathit{false}, \mathit{IN})\ \wedge\ R[p] = (\mathit{false}, \mathit{IN})$ holds continually after $p.10$ is executed. Note that, by (I115), this implies that process q is not blocked at $q.11$ and $q.12$ because $q \neq p$. Because $p@\{11,12\}\ \wedge\ P[p] = (\mathit{false}, \mathit{IN})\ \wedge\ R[p] = (\mathit{false}, \mathit{IN})$ continues to hold, by (I115), $p@\{11,12\}\ \wedge\ P[p] = (\mathit{false}, \mathit{IN})\ \wedge\ R[p] = (\mathit{false}, \mathit{IN})\ \wedge\ Q = p$ continues to hold. Hence, because process q is nonfaulty and does not become blocked at $q.11$ and $q.12$, statement $q.15$ is eventually executed when $p@\{11,12\}\ \wedge\ P[p] = (\mathit{false}, \mathit{IN})\ \wedge\ R[p] = (\mathit{false}, \mathit{IN})\ \wedge\ Q = p$ holds, establishing $q@16\ \wedge\ q.v = Q\ \wedge\ R[p] = (\mathit{false}, \mathit{IN})\ \wedge\ Q = p$. Statement $q.16$ is then eventually executed, establishing $R[p] = (\mathit{true}, \mathit{IN})$. Thus, $p@\{11,12\}\ \wedge\ (P[p] \neq$

$(false, IN) \ \vee \ R[p] \neq (false, IN))$ holds at some state after $p.10$ is executed.

To conclude the proof, note that once $p@\{11, 12\} \ \wedge \ (P[p] \neq (false, IN) \ \vee \ R[p] \neq (false, IN))$ holds, by (U7), $p@13$ eventually holds, because p is nonfaulty. Thus, Starvation-Freedom holds for the program of Figure 6.9. $\square$

Appendix C

Correctness Proofs for Algorithms in Chapter 7

In this appendix, we provide formal correctness proofs for the renaming algorithms presented in Chapter 7. Each algorithm is considered in a separate section.

C.1 Correctness Proof for Algorithm in Figure 7.6

In accordance with the problem definition given in Section 7.1, we assume (I118) and are required to prove (I119) and that our algorithm satisfies the wait-freedom property.

invariant $|\{p :: p@\{1..11\}| \leq k$ (I118)

invariant $(p \neq q \ \wedge \ p@9 \ \wedge \ q@9 \Rightarrow p.name \neq q.name) \ \wedge$

$$(p@9 \Rightarrow 0 \leq p.name < k(k+1)/2) \quad \text{(I119)}$$

Wait-Freedom: Every nonfaulty process that leaves line 0 eventually reaches line 9, and that every nonfaulty process that leaves line 9 eventually reaches line 0.

The proofs of the following invariants are straightforward, and are therefore omitted. In particular, (I120) through (I126) follow directly from the program text, (I127) is proved using (I121), (I128) is proved using (I126), and (I125) is proved using (I128). Finally, the proof of (I129) uses (I128), and the proof of (I130) uses (I129).

invariant $p@\{0..11\}$ (I120)

invariant $p@\{5,10\} \Rightarrow p.h > 0$ (I121)

invariant $p@9 \Rightarrow p.name = (p.i)k - (p.i)(p.i-1)/2 + p.j$ (I122)

invariant $(\forall r,c,n : 0 \le r < k-1 \ \wedge\ 0 \le c < k-1 \ \wedge\ r+c < k-1 \ \wedge$

$0 \le n \le k-r-c :: Y[r,c,n] \in \{\perp\} \cup \{0..N-1\})$ (I123)

invariant $p@\{2..7\} \Rightarrow \neg p.moved$ (I124)

invariant $p@2 \Rightarrow p.h < k - p.i - p.j$ (I125)

invariant $p@\{1..7\} \Rightarrow p.i + p.j < k-1$ (I126)

invariant $p.i \ge 0 \ \wedge\ p.j \ge 0 \ \wedge\ p.h \ge 0$ (I127)

invariant $p.i + p.j \le k-1$ (I128)

invariant $p@\{3..7\} \Rightarrow p.h \le k - p.i - p.j$ (I129)

invariant $p@\{8..11\} \Rightarrow p.h \le k - p.i - p.j + 1$ (I130)

The following invariant shows that if the nth Y component in building block (r,c) is set, then some process is accessing building block (r,c) at, or beyond, component n.

invariant $(r \ge 0 \ \wedge\ c \ge 0 \ \wedge\ r+c < k-1 \ \wedge\ n \ge 0 \ \wedge\ n < k-r-c \ \wedge$

$Y[r,c,n] = q) \ \Rightarrow\ (q.i = r \ \wedge\ q.j = c \ \wedge\ \neg q.moved \ \wedge$

$$((q@\{3..6, 8..11\} \ \wedge \ q.h > n) \ \vee \ (q@\{6, 11\} \ \wedge \ q.h = n))) \qquad \text{(I131)}$$

Proof: Assume $r \geq 0 \ \wedge \ c \geq 0 \ \wedge \ r + c < k - 1 \ \wedge \ n \geq 0 \ \wedge \ n < k - r - c$. This implies that $Y[r, c, n] = \perp$ holds initially, so (I131) holds initially. Only statement $q.4$ can establish the antecedent, and it does so only if executed when $q@4 \ \wedge \ q.i = r \ \wedge \ q.j = c \ \wedge \ q.h = n$ holds. By (I124), $\neg q.moved$ also holds in this case. Therefore, because $n < k - r - c$, $q.4$ establishes $q.i = r \ \wedge \ q.j = c \ \wedge \ \neg q.moved \ \wedge \ q@3 \ \wedge \ q.h > n$, thereby establishing the consequent.

No statement modifies $q.i$, $q.j$, or $q.moved$ while the consequent holds. It remains to consider statements that might falsify $((q@\{3..6, 8..11\} \wedge q.h > n) \vee (q@\{6, 11\} \wedge q.h = n))$ while the consequent holds. First, observe that any statement that falsifies the second disjunct while the consequent holds also falsifies the antecedent.

If the first disjunct holds, then $q.h > 0$. Therefore, statement $q.3$ establishes $q@\{4, 5\}$, and does not modify $q.h$. By (I124), statement $q.4$ establishes $q@3$ or $q@8$ and increases $q.h$, so $q.4$ does not falsify first disjunct. If $q.h > n + 1$, then $q@\{5, 6\} \ \wedge \ q.h > n$ holds after $q.5$ is executed. If $q.h = n + 1$, then because $Y[r, c, n] = q$, statement $q.5$ establishes $q@6 \wedge q.h = n$, thereby establishing the second disjunct listed above. Similarly, $(q@\{10..11\} \ \wedge \ q.h > n) \ \vee \ (q@11 \ \wedge \ q.h = n)$ holds after $q.10$ is executed. Because $q.h > n$ and $n \geq 0$, statement $q.6$ establishes $q@5 \ \wedge \ q.h > n$. Similarly, statement $q.11$ establishes $q@10 \wedge q.h > n$. Finally, statements $q.8$ and $q.9$ do not modify $q.h$, $q.8$ establishes $q@9$, and because $\neg q.moved$ and $q.h > n$ holds, statement $q.9$ establishes $q@10$. Thus, no statement falsifies the consequent while the antecedent holds. $\square$

Definitions. For convenience, we define the following predicates. Intuitively, $MOD(q, r, c, n)$ holds if process q is about to modify $Y[r, c, n]$; $SET(q, r, c, n)$ holds if process q has just set $Y[r, c, n]$ to q and has not yet reset it; and $EN(q, r, c)$ holds if process q will access (or has already accessed) a building in the subgrid whose top-left corner is at (r, c).

$$
\begin{aligned}
MOD(q, r, c, n) &\equiv\ q.i = r \ \wedge\ q.j = c \ \wedge\ q.h = n \ \wedge\ q@\{4, 6, 11\} \\
SET(q, r, c, n) &\equiv\ q.i = r \ \wedge\ q.j = c \ \wedge \\
&\qquad ((q@\{3, 5, 8..10\} \ \wedge\ q.h = n + 1) \ \vee\ (q@\{6, 11\} \ \wedge\ q.h = n)) \\
EN(q, r, c) &\equiv\ q.j \geq c \ \wedge\ ((q.i \geq r \ \wedge\ q@\{1..11\}) \ \vee\ (q.i = r - 1 \ \wedge \\
&\qquad ((q@\{2..4\} \ \wedge\ X[r - 1, q.j] \neq q) \ \vee\ q@\{5..7\}))) \quad \Box
\end{aligned}
$$

The following three invariants follow easily from the definitions above. The proof of (I133) uses (I132).

invariant $(SET(p, r, c, n) \ \vee\ MOD(p, r, c, n)) \Rightarrow$

$$(\forall m : m \neq n :: \neg SET(p, r, c, m) \ \wedge\ \neg MOD(p, r, c, m)) \quad \text{(I132)}$$

invariant $p@2 \ \wedge\ EN(p, r, c) \Rightarrow |\{n :: (\exists q :: SET(q, r, c, n) \ \vee$

$$MOD(q, r, c, n))\}| < |\{q :: EN(q, r, c)\}| \quad \text{(I133)}$$

invariant $EN(p, r, c) \Rightarrow EN(p, r, c - 1) \ \wedge\ EN(p, r - 1, c)$ (I134)

The next invariant implies that at most $k - r - c$ processes access building blocks in the sub-grid whose top left corner is at building block (r, c). In particular, this implies that at most one process at a time occupies each grid position that is $k - 1$ steps from the position origin.

invariant $r \geq 0 \ \wedge \ c \geq 0 \ \wedge \ r + c \leq k - 1 \Rightarrow (|\{p :: EN(p, r, c)\}| \leq k - r - c)$ (I135)

Proof: Initially, $(\forall p :: p@0)$ holds, so (I135) holds. First, observe that (I118) implies that if $r = 0 \ \wedge \ c = 0$, then (I135) holds. Henceforth, assume that $r \geq 0 \ \wedge \ c \geq 0 \ \wedge \ r + c \leq k - 1 \ \wedge \ r + c > 0$. (I135) can be falsified only by establishing $EN(q, r, c)$ for some process q. By the definition of EN, this can be achieved only by modifying $q.i$, $q.j$ or $X[r-1, q.j]$, or by establishing $q@\{1..11\}$ or $q@\{2..4\}$ or $q@\{5..7\}$. The statements to check are therefore $q.0$, $q.2$, $q.3$, $q.7$, and $p.1$, where p is any process.

Because $r + c > 0$, statement $q.0$ does not establish $EN(q, r, c)$. Statement $q.3$ potentially establishes $EN(q, r, c)$ only by establishing $q@\{5..7\}$. Thus, $EN(q, r, c)$ holds after $q.3$ is executed only if $q.j \geq c \ \wedge \ q.i = r - 1 \ \wedge \ q@3 \ \wedge \ X[q.i, q.j] \neq q$ holds before, in which case $EN(q, r, c)$ already holds. Statement $q.7$ establishes $q@\{1, 8\}$ and could therefore establish $EN(q, r, c)$ only by establishing $q.j \geq c \ \wedge \ q.i \geq r$. However, it does so only if executed when $q.j \geq c \ \wedge \ q.i = r - 1 \ \wedge \ q@7$ holds, in which case $EN(q, r, c)$ already holds. It remains to consider statement $p.1$, where p is any process, and statement $q.2$.

If $p = q \ \wedge \ q.i \geq r$, then $p.1$ does not establish $EN(q, r, c)$ because it does not modify $q.i$ or $q.j$ or establish $q@\{1..11\}$. If $p = q \ \wedge \ q.i < r$, then by (I127), statement $p.1$ establishes $q@2 \ \wedge \ (q.i \neq r - 1 \ \vee \ X[r-1, q.j] = q)$ and therefore does not establish $EN(q, r, c)$. If $p \neq q$, then statement $p.1$ can establish $EN(q, r, c)$ only by establishing $X[r-1, q.j] \neq q$ while $q.j \geq c \wedge q.i = r - 1 \wedge q@\{2..4\} \wedge X[r-1, q.j] = q$ holds. However, it does so only if executed when $p@1 \ \wedge \ p.i = r - 1 \ \wedge \ p.j \geq c$. These assertions imply $EN(q, r-1, c) \ \wedge \ \neg EN(q, r, c) \ \wedge \ EN(p, r-1, c) \ \wedge \ \neg EN(p, r, c)$. Also, $q.i = r - 1 \ \wedge$ (I127) implies that $r - 1 \geq 0$ and $r + c \leq k - 1$ implies that $r - 1 + c \leq k - 1$. Therefore, because

$(\mathrm{I}135)^{r,c}_{r-1,c}$ holds before $p.1$ is executed, it follows that $|\{p :: EN(p, r-1, c)\}| \leq k - r - c + 1$ holds before $p.1$ is executed. By (I134), this implies that $|\{p :: EN(p, r, c)\}| \leq k - r - c - 1$ holds before $p.1$ is executed (because $p \neq q$ and $EN(q, r-1, c) \wedge \neg EN(q, r, c) \wedge EN(p, r-1, c) \wedge \neg EN(p, r, c)$ holds), so $p.1$ does not falsify (I135).

Statement $q.2$ can establish $EN(q, r, c)$ only if executed when $q@2 \wedge q.i \geq r \wedge q.j = c-1 \wedge Y[q.i, c-1, q.h] \neq \perp$ holds. By (I127) and (I125), this implies that $q.i \geq 0 \wedge c-1 \geq 0 \wedge q.h \geq 0 \wedge q.h < k - q.i - (c-1)$ holds before $q.2$ is executed. Thus, by (I123), (I126), and $(\mathrm{I}131)^{r,c,n,q}_{q.i,c-1,q.h,s}$, it follows that $(\exists s : s \neq q :: EN(s, r, c-1) \wedge \neg EN(s, r, c))$ holds. (Note that $q@2 \wedge s@\{3..6, 8..11\}$ implies that $s \neq q$.) Also, $EN(q, r, c-1) \wedge \neg EN(q, r, c)$ holds. Thus, as above, (I127), (I134), and $(\mathrm{I}135)^{r,c}_{r,c-1}$ imply that $|\{p :: EN(p, r, c)\}| \leq k - r - c - 1$ holds before $q.2$ is executed, so $q.2$ does not falsify (I135). $\Box$

The following invariant implies that, while process p is executing the loop at line 2 and $X[p.i, p.j] = p$ still holds, for each component of Y that p has already read, either that component is not set, or some process has just written that component and has not yet cleared it.

$$\textbf{invariant } (r \geq 0 \ \wedge \ c \geq 0 \ \wedge \ r + c < k - 1 \ \wedge \ p@2 \ \wedge \ p.i = r \ \wedge \ p.j = c \ \wedge \ X[r, c] = p) \Rightarrow (\forall n : 0 \leq n < p.h :: Y[r, c, n] = \perp \ \vee \ (\exists q : q \neq p :: Y[r, c, n] = q \ \wedge \ \neg q.moved \ \wedge \ SET(q, r, c, n))) \quad (\mathrm{I}136)$$

Proof: Assume $r \geq 0 \wedge c \geq 0 \wedge r + c < k - 1$. Initially, $p@0$ holds, so (I136) holds. Only statement $p.1$ establishes $p@2$. Statement $p.1$ also establishes $p.h = 0$, so the consequent holds vacuously after $p.1$ is executed. No statement modifies $p.i$ while $p@2$ holds, and only statement $p.2$ modifies $p.j$ while $p@2$ holds. However, if $p.2$ modifies $p.j$, then it also establishes $p.moved$ and terminates the loop, thereby falsifying the antecedent. Only statement $p.1$ can establish $X[r,c] = p$, and as shown above, the consequent holds after the execution of $p.1$.

We now show that no statement falsifies the consequent while the antecedent holds. Statements of process p other than $p.2$ are not enabled while the antecedent holds. Statement $p.2$ can affect the consequent only by increasing $p.h$. However, $p.2$ does not falsify the consequent in this case, because it increments $p.h$ only if $Y[r,c,p.h] = \perp$. We now consider steps of process s, where $s \neq p$. Statements of process s can falsify the consequent only by modifying $Y[r,c,n]$ or by falsifying $\neg s.moved \wedge SET(s,r,c,n)$ for some $n < p.h$ while $Y[r,c,n] = s$ holds.

Only statements $s.4$, $s.6$, and $s.11$ modify $Y[r,c,n]$. Statements $s.6$ and $s.11$ establish $Y[s.i, s.j, s.h] = \perp$ and therefore do not falsify the consequent. Statement $s.4$ modifies $Y[r,c,n]$ only if $s@4 \wedge s.i = r \wedge s.j = c \wedge s.h = n$. By (I124), $\neg s.moved$ holds in this case. Therefore, after $s.4$ is executed in this case, $Y[r,c,n] = s \wedge s.i = r \wedge s.j = c \wedge \neg s.moved \wedge s@\{3,8\} \wedge s.h = n+1$ holds, which implies $SET(s,r,c,n)$. Thus, $s.4$ does not falsify the consequent by modifying $Y[r,c,n]$.

We now consider statements that potentially falsify $\neg s.moved \wedge SET(s,r,c,n)$ for some process s and for some $n < p.h$ while $Y[r,c,n] = s$ holds. No statement modifies $s.i$, $s.j$, or $s.moved$ while $SET(s,r,c,n)$ holds. It remains to consider statements that potentially falsify $((s@\{3,5,8..10\} \wedge s.h = n+1) \vee (s@\{6,11\} \wedge s.h = n))$ while $Y[r,c,n] = s \wedge \neg s.moved \wedge SET(s,r,c,n)$ holds. First, observe that if $s.6$ or $s.11$ falsifies the second disjunct while the antecedent and consequent both hold, then it also establishes $Y[r, c, n] = \perp$, thereby preserving the consequent. If the first disjunct holds, then $s.h > 0$. Thus, because the antecedent implies that $X[r, c] \neq s$ (recall that $s \neq p$), statement $s.3$ establishes $s@5$ and does not modify $s.h$. Also, because $Y[r, c, n] = s$, executing $s.5$ establishes $s@6 \ \wedge \ s.h = n$, thereby establishing the second disjunct above. Similarly, statement $s.10$ establishes $s@11 \ \wedge \ s.h = n$ if executed while the antecedent and consequent both hold. Finally, statements $s.8$ and $s.9$ do not modify $s.h$, $s.8$ establishes $s@9$, and because $\neg s.moved$ and $s.h = n + 1$ holds, statement $s.9$ establishes $s@10$. $\Box$

The following invariant implies that while process p is executing the loop at lines 3 to 4 and $X[p.i, p.j] = p$ still holds, one of the Y-components that p has set is not overwritten by any other process.

invariant $r \geq 0 \ \wedge c \geq 0 \ \wedge \ r + c < k - 1 \ \wedge$

$$p@\{3..4\} \ \wedge \ p.i = r \ \wedge \ p.j = c \ \wedge \ X[r, c] = p \Rightarrow$$

$$(\exists n :: (\forall q : q \neq p :: \neg MOD(q, r, c, n)) \ \wedge$$

$$((Y[r, c, n] = p \ \wedge \ 0 \leq n \ \wedge \ n < p.h) \ \vee$$

$$(Y[r, c, n] = \perp \ \wedge \ p.h \leq n \ \wedge \ n < k - r - c)) \quad \text{(I137)}$$

Proof: Assume $r \geq 0 \ \wedge \ c \geq 0 \ \wedge \ r + c < k - 1$. Initially, $p@0$ holds, so (I137) holds. Observe that no statement modifies $p.i$ or $p.j$ or establishes $X[r, c] = p$ while $p@\{3..4\}$ holds. Thus, only statements that establish $p@\{3..4\}$ can establish the antecedent. By (I126), statement $p.1$ establishes $p.h < k - p.i - p.j \ \wedge \ \neg p.moved$, so $p.1$ does not establish $p@\{3..4\}$. After statement $p.5$ or statement $p.6$ is executed, $p@\{5..7\}$ holds. Statement $p.7$ establishes $p.moved$, thereby terminating the loop and establishing $p@\{1, 8\}$. The following assertions imply that if statement $p.2$ establishes the antecedent, then the consequent holds afterwards.

$$\{p@2 \ \wedge \ (p.i \neq r \ \vee \ p.j \neq c \ \vee \ Y[p.i, p.j, p.h] \neq \perp \ \vee \ X[r,c] \neq p)\} \ p.2$$
$$\{(p.i \neq r \ \vee \ p.j \neq c) \ \vee \ (p.moved \ \wedge \ p@\{1,8\}) \ \vee \ X[r,c] \neq p\}$$

, $p.2$ does not modify $p.i$ or X; if $p.2$ modifies $p.j$ or

if $Y[p.i, p.j, p.h] \neq \perp$, then $p.2$ establishes $p.moved \ \wedge \ p@\{1,8\}$.

$$p@2 \ \wedge \ p.i = r \ \wedge \ p.j = c \ \wedge \ Y[p.i, p.j, p.h] = \perp \ \wedge \ X[r,c] = p \ \wedge$$
$$p.h + 1 > k - r - c \Rightarrow \mathit{false}$$

, by (I125).

$$\{p@2 \ \wedge \ p.i = r \ \wedge \ p.j = c \ \wedge \ Y[p.i, p.j, p.h] = \perp \ \wedge \ X[r,c] = p \ \wedge$$
$$p.h + 1 < k - r - c\} \ p.2 \ \{p@2\}$$

, by (I124), the precondition implies $\neg p.moved \ \wedge \ p.h + 1 < k - p.i - p.j$,

so the loop does not terminate.

$$p@2 \ \wedge \ p.i = r \ \wedge \ p.j = c \ \wedge \ Y[p.i, p.j, p.h] = \perp \ \wedge \ X[r,c] = p \ \wedge \ p.h + 1 = k - r - c$$

$$\Rightarrow \ p@2 \ \wedge \ p.i = r \ \wedge \ p.j = c \ \wedge \ Y[r, c, p.h] = \perp \ \wedge \ X[r,c] = p \ \wedge$$
$$p.h + 1 = k - r - c \ \wedge \ |\{n :: (\exists q :: SET(q,r,c,n) \ \vee \ MOD(q,r,c,n))\}| <$$
$$|\{q :: EN(q,r,c)\}|$$

, by (I133) and the definition of $EN(p,r,c)$.

$$\Rightarrow \ p@2 \ \wedge \ p.i = r \ \wedge \ p.j = c \ \wedge \ Y[r, c, p.h] = \perp \ \wedge \ X[r,c] = p \ \wedge$$
$$p.h + 1 = k - r - c \ \wedge$$
$$|\{n :: (\exists q :: SET(q,r,c,n) \ \vee \ MOD(q,r,c,n))\}| < k - r - c$$

, by (I135).

$$\Rightarrow \ p@2 \ \wedge \ p.i = r \ \wedge \ p.j = c \ \wedge \ Y[r, c, p.h] = \perp \ \wedge \ X[r,c] = p \ \wedge$$
$$p.h + 1 = k - r - c \ \wedge \ (\exists n : 0 \leq n < k - r - c :: (\forall q :: \neg SET(q,r,c,n) \ \wedge$$
$$\neg MOD(q,r,c,n)))$$

, by the pigeonhole principle.

$\Rightarrow\ p@2\ \wedge\ p.i = r\ \wedge\ p.j = c\ \wedge\ Y[r, c, p.h] = \perp\ \wedge\ X[r, c] = p\ \wedge$

$p.h + 1 = k - r - c\ \wedge\ (\exists n : 0 \leq n < k - r - c :: Y[r, c, n] = \perp\ \wedge$

$(\forall q :: \neg SET(q, r, c, n)\ \wedge\ \neg MOD(q, r, c, n)))$, by (I136).

$\{p@2\ \wedge\ p.i = r\ \wedge\ p.j = c\ \wedge\ Y[r, c, p.h] = \perp\ \wedge\ X[r, c] = p\ \wedge$

$p.h + 1 = k - r - c \wedge$ (I133) $\wedge$ (I135) $\wedge$ (I136)$\}$ $p.2$

$\{(\exists n :: Y[r, c, n] = \perp\ \wedge\ p.h \leq n\ \wedge\ n < k - r - c\ \wedge$

$(\forall q : q \neq p :: \neg MOD(q, r, c, n)))\}$

, by the preceding derivation and the program text; note that $p.2$ establishes $p.h = 0$ in this case, and does not modify Y or establish $MOD(q, r, c, n)$ for any q. Also observe that the postcondition implies the consequent of (I137). □

The following invariant implies that, if process p is in its working section while occupying an interior building block (r, c), then no other process is in its working section at that building block.

invariant $r \geq 0\ \wedge\ c \geq 0\ \wedge\ r + c < k - 1\ \wedge\ p.i = r\ \wedge\ p.j = c\ \wedge$

$((p@4\ \wedge\ p.h \geq k - r - c)\ \vee\ p@\{8..9\}) \Rightarrow$

$(\exists n : 0 \leq n < k - r - c :: Y[r, c, n] = p\ \wedge\ (\forall q : q \neq p :: q.i \neq p.i\ \vee\ q.j \neq p.j$

$\vee\ q@\{0..1\}\ \vee\ (q@2\ \wedge\ q.h \leq n)\ \vee\ ((q@\{2..3\}\ \vee$

$(q@4\ \wedge\ q.h < k - r - c\ \wedge\ q.h \neq n))\ \wedge\ X[r, c] \neq q)\ \vee$

$q@\{5, 7, 10\}\ \vee\ (q@\{6, 11\}\ \wedge\ q.h \neq n)))$ (I138)

Proof: Assume that $r \geq 0 \wedge c \geq 0 \wedge r+c < k-1$. Initially, $p@0$ holds, so (I138) holds. The antecedent of (I138) is only established by modifying $p.i$, $p.j$, or $p.h$, or by establishing $p@4$ or $p@\{8..9\}$. Therefore, statements $p.6$, $p.8$, $p.9$, and $p.11$ do not establish the antecedent. After statements $p.0$, $p.1$, $p.5$, and $p.10$, the antecedent does not hold because $\neg p@\{4, 8..9\}$ holds. ((I124) implies that $p.1$ establishes $p@2$.) Statement $p.7$ establishes $p@1 \ \vee \ (p@8 \ \wedge \ p.i + p.j \geq k-1)$, and therefore does not establish the antecedent.

If $p@2 \wedge Y[p.i, p.j, p.h] = \bot$ holds before $p.2$ is executed, then, by (I124) and (I125), $p@\{2, 3\}$ holds afterwards. If $p@2 \ \wedge \ Y[p.i, p.j, p.h] \neq \bot$ holds before $p.2$ is executed, then $p@1 \ \vee \ (p@8 \ \wedge \ p.i + p.j \geq k-1)$ holds afterwards. Thus, $p.2$ does not establish the antecedent. The remaining statements to check are $p.3$ and $p.4$.

By (I124) and (I129), $\neg p.moved \ \wedge \ p.h \leq k - p.i - p.j$ holds before statement $p.4$ is executed. Therefore, the antecedent holds after $p.4$ is executed only if $p.i = r \ \wedge \ p.j = c \ \wedge \ p@4 \ \wedge \ p.h = k - p.i - p.j$ holds before, in which case the antecedent already holds.

Statement $p.3$ establishes the antecedent of (I138) only if executed when $p@3 \ \wedge \ p.i = r \ \wedge \ p.j = c \ \wedge \ X[p.i, p.j] = p \ \wedge \ p.h \geq k - r - c$ holds. By (I129) and $(\text{I137})^{r,c}_{p.i,p.j}$, this implies that the following assertion holds before $p.3$ is executed.

$$X[p.i, p.j] = p \ \wedge \ (\exists n : 0 \leq n < k - r - c :: Y[r, c, n] = p \ \wedge \ (\forall q : q \neq p :: X[p.i, p.j] \neq q \ \wedge \ \neg MOD(q, p.i, p.j, n))) \quad \text{(A9)}$$

(A9) $\wedge$ (I120) $\wedge$ $(\text{I138})^{p,q,r,c}_{q,p,q.i,q.j}$ implies the consequent of (I138). To see this, suppose that some $q \neq p$ does not satisfy the universal quantifier in the consequent of (I138). Then (A9) $\wedge$ (I120) implies that $q.i = p.i \ \wedge \ q.j = p.j \ \wedge \ ((q@4 \ \wedge \ q.h \geq k - r - c) \ \vee \ q@\{8..9\})$ holds (because $X[p.i, p.j] \neq q \ \wedge \ \neg MOD(q, p.i, p.j, n)$ holds). In this case, $(\mathrm{I138})^{p,q,r,c}_{q,p,q.i,q.j} \wedge p@3$ implies $X[p.i, p.j] \neq p$, a contradiction. Statement $p.3$ does not falsify the consequent, so the consequent holds after $p.3$ is executed in this case.

We now consider statements that potentially falsify the consequent while the antecedent holds. First, observe that no statement modifies $p.i$, $p.j$, or $Y[r, c, n]$ while the antecedent and consequent both hold. Only statements $q.0$, $q.2$, and $q.7$ potentially falsify the consequent by modifying $q.i$ or $q.j$ for some process q. However, if any of these statements modifies $q.i$ or $q.j$, then $q@1 \vee (q@8 \wedge q.i+q.j \geq k-1)$ holds afterwards. The latter disjunct implies that $q.i \neq p.i \vee q.j \neq p.j$ holds because the antecedent implies that $p.i+p.j < k-1$. Thus, no statement that modifies $q.i$ or $q.j$ for any process q falsifies the consequent while the antecedent holds. It therefore remains to consider statements that falsify (A10) below, for some $q \neq p$, while $p.i = r \wedge p.j = c \wedge 0 \leq n < k-r-c \wedge Y[r, c, n] = p \wedge q.i = p.i \wedge q.j = p.j$ holds.

$$
\begin{aligned}
&q@\{0..1\} \ \vee \ (q@2 \ \wedge \ q.h \leq n) \ \vee \ ((q@\{2..3\} \ \vee \\
&(q@4 \ \wedge \ q.h < k - r - c \ \wedge \ q.h \neq n)) \ \wedge \ X[r, c] \neq q) \ \vee \\
&\qquad\qquad\qquad q@\{5, 7, 10\} \ \vee \ (q@\{6, 11\} \ \wedge \ q.h \neq n)
\end{aligned} \tag{A10}
$$

Only statement $q.1$ falsifies $q@\{0..1\}$, and $q.1$ establishes $q@2 \ \wedge \ q.h \leq n$.

Only statements $q.2$ and $q.3$ falsify $q@2 \ \wedge \ q.h \leq n$ or $q@\{2..3\}$. As shown above, if $q.2$ modifies $q.j$, then it does not falsify the consequent while the antecedent holds. Also, statement $q.2$ does not falsify $q@2 \ \wedge \ q.h \leq n$ by incrementing $q.h$ because $Y[q.i, q.j, n] \neq q$. By (I124) and (I125), statement $q.2$ does not falsify $q@\{2..3\}$ if it increments $q.h$. If statement $q.3$ falsifies $q@\{2..3\}$ while $X[r, c] \neq q$ holds, then $q.3$ establishes $q@\{5, 7\}$. (Recall that $p.i = c \ \wedge \ p.j = r \ \wedge \ q.i = p.i \ \wedge \ q.j = p.j$.)

If statement $q.4$ falsifies $q@4 \ \wedge \ q.h < k-r-c \ \wedge \ q.h \neq n$, then by (I124), $q.4$ establishes $q@3$. No statement falsifies $X[r,c] \neq q$ while $q@\{2..4\}$ holds. Because $q.i = r \ \wedge \ q.j = c \ \wedge \ Y[r,c,n] \neq q$ holds, $(q@6 \ \wedge \ q.h \neq n) \ \vee \ q@\{5,7\}$ holds after the execution of $q.5$. As shown above, statement $q.7$ does not falsify the consequent while the antecedent holds. Statement $q.10$ establishes $q@\{0,10\} \ \vee \ (q@11 \ \wedge \ q.h \neq n)$ (because $q.i = r \wedge q.j = c \wedge Y[r,c,n] = p$). Finally, statement $q.6$ establishes $q@\{5,7\}$ and statement $q.11$ establishes $q@\{0,10\}$. Thus, the consequent is not falsified while the antecedent holds.$\Box$

The following invariant implies that, if two distinct processes are in their working sections concurrently, then they occupy different grid positions. We ues this fact later to show that distinct processes do not hold the same name concurrently.

invariant $p \neq q \ \wedge \ p@\{8..9\} \ \wedge \ q@\{8..9\} \Rightarrow p.i \neq q.i \ \vee \ q.j \neq q.j$ (I139)

Proof: If $p \neq q \ \wedge \ p@\{8..9\} \ \wedge \ p.i + p.j < k-1$ holds, then by (I127) and $(\text{I138})^{r,c}_{p.i,p.j}$, $q.i \neq p.i \ \vee \ q.j \neq p.j \ \vee \ \neg q@\{8..9\}$ holds, so (I139) holds. If $p \neq q \ \wedge \ p@\{8..9\} \ \wedge \ p.i + p.j \geq k-1$ holds, then by (I127), (I128), and $(\text{I135})^{r,c}_{p.i,p.j}$, it follows that $|\{s :: EN(s, p.i, p.j)\}| \leq 1$. By the definition of EN, the antecedent implies $EN(p, p.i, p.j)$. Therefore $\neg EN(q, p.i, p.j)$ holds, which implies that the consequent holds. $\Box$

Claim 19: Let c, d, c', and d' be nonnegative integers satisfying $(c \neq c' \vee d \neq d') \wedge (c+d \leq k-1) \ \wedge \ (c'+d' \leq k-1)$. Then, $ck - c(c-1)/2 + d \neq c'k - c'(c'-1)/2 + d'$.

Proof: The claim is straightforward if $c = c'$, so assume that $c \neq c'$. Without loss of generality assume that $c < c'$. Then,

$$ck - c(c-1)/2 + d \leq ck - c(c-1)/2 + k - 1 - c \qquad , d \leq k-1-c.$$

$$
\begin{aligned}
&= ck - c^2/2 - c/2 + k - 1 \\
&< (c+1)(k - c/2) \\
&\leq c'k - c'(c'-1)/2 && , c+1 \leq c'. \\
&\leq c'k - c'(c'-1)/2 + d' && , d' \text{ is nonnegative.} \quad \square
\end{aligned}
$$

Claim 20: Let c and d be nonnegative integers satisfying $c + d \leq k - 1$. Then $0 \leq ck - c(c-1)/2 + d < k(k+1)/2$.

Proof: It follows from the statement of the claim that $c \leq k - 1$. Thus, $k - (c-1)/2 > 0$. Also, $c \geq 0$ and $d \geq 0$. Thus, $ck - c(c-1)/2 + d \geq 0$. To see that $ck - c(c-1)/2 + d < k(k+1)/2$, consider the following derivation.

$$
\begin{aligned}
ck - c(c-1)/2 + d &\leq ck - c(c-1)/2 + d(d+1)/2 && , d \geq 0. \\
&\leq ck - c(c-1)/2 + (k-1-c)(k-c)/2 && , d \leq k-1-c. \\
&= c + k(k-1)/2 \\
&\leq k - 1 + k(k-1)/2 && , c \leq k-1. \\
&< k(k+1)/2 && \square
\end{aligned}
$$

The next two invariants show that distinct processes in their working sections hold distinct names from $\{0, ..., k(k+1)/2-1\}$. The first follows easily from (I122), (I127), (I128), (I139), and Claim 19. The second is easily proved using (I122), (I127), (I128), and Claim 20.

invariant $p \neq q \ \wedge \ p@9 \ \wedge \ q@9 \Rightarrow p.name \neq q.name$ (I140)

invariant $p@9 \Rightarrow 0 \leq p.name < k(k+1)/2$ (I141)

(I140) and (I141) imply that the algorithm in Figure 7.6 correctly implements $(k(k+1)/2)$-renaming. To see that the wait-freedom requirement is satisfied, we consider all the loops in the algorithm in Figure 7.6. By (I127), the loop at line 2 clearly terminates after at most k iterations. Similarly, by (I127), (I129), and (I130), the loop at lines 5 and 6 and the loop at lines 10 to 11 both terminate after at most $k+1$ iterations. Also, note that if the loop at lines 5 to 6 is executed, then statement 7 establishes $p.moved$, so the loop at lines 3 to 7 terminates. Thus, the loop at lines 5 to 6 is executed at most once per execution of the loop at lines 3 to 7. To see that the loop at lines 3 to 7 terminates, consider statement $p.3$. If $X[p.i, p.j] \neq p$ holds before statement $p.3$ is executed, then statement $p.7$ establishes $p.moved$, so the loop terminates. Otherwise, $p.h$ is incremented when statement $p.4$ is executed. Because of the loop condition $p.h \leq k - p.i - p.j$, by (I127), the loop at lines 3 to 7 is executed at most k times. Finally, the loop at lines 1 to 7 executes at most $k-1$ times. To see this, observe that, by (I124), the loop terminates unless some statement establishes $p.moved$. Only statements $p.2$ and $p.7$ establish $p.moved$, and if they do so, they increment either $p.i$ or $p.j$. Thus, because of the loop condition $p.i + p.j < k - 1$, (I127) implies that the loop terminates after at most $k-1$ executions. Thus, we have the following result.

Theorem 21: Using *read* and *write*, wait-free, long-lived $(k(k+1)/2)$-renaming can be implemented so that the worst-case time complexity of acquiring and releasing a name once is $\Theta(k^2)$, and the space complexity is $\Theta(k^3)$. $\square$

C.2 Correctness Proof for Algorithm in Figure 7.7

In accordance with the problem specification, we assume the following invariant.

invariant $|\{p :: p@\{1..3\}\}| \leq k$ (I142)

The following invariants follow directly from the program text in Figure 7.7, and are stated without proof.

invariant $p@3 \Rightarrow p.name = b(p.h) + p.v$ (I143)

invariant $p.h \geq 0$ (I144)

invariant $p@\{2..3\} \Rightarrow 0 \leq p.v < b$ (I145)

Correctness proofs are given below for the remaining invariants. Although each of the following two assertions is an invariant in its own right, it is convenient to prove that their conjunction is an invariant because this way we may inductively assume that both hold before any statement execution. These assertions show that two processes do not concurrently "hold" the same bit and that for each set bit, some process r holds that bit.

$q \neq p \;\wedge\; q@\{2..3\} \;\wedge\; p@\{2..3\} \;\wedge\; 0 \leq p.h < \lceil k/b \rceil \Rightarrow q.h \neq p.h \;\vee\; q.v \neq p.v$ (A11)

$0 \leq i < \lceil k/b \rceil \;\wedge\; 0 \leq j < b \Rightarrow (X[i][j] \equiv (\exists r :: r@\{2,3\} \;\wedge\; r.h = i \;\wedge\; r.v = j))$ (A12)

invariant (A11) $\wedge$ (A12) (I146)

Proof: Initially $(\forall p :: p@0) \;\wedge\; \neg X[i][j]$ holds, so (I146) holds. We first consider statements that potentially falsify (A11). Assume that $q \neq p$. By (I144), only $p.0$ can establish $0 \leq p.h < \lceil k/b \rceil$, and the antecedent does not hold after $p.0$ is executed. Therefore, by symmetry, we need only consider statements that may establish $q@\{2..3\}$ or modify $q.h$ or $q.v$. The statements to check are $q.0$ and $q.1$. The antecedent does not hold after $q.0$ is executed. To show that statement $q.1$ does not falsify (A11), we consider the following three cases.

$\{q@1 \ \wedge \ (\forall n : 0 \leq n < b :: X[q.h][n])\} \ q.1 \ \{q@1\}$

, $q.1$ assigns $q.v = b$ so loop does not terminate.

$\{q@1 \ \wedge \ (\exists n : 0 \leq n < b :: \neg X[q.h][n]) \ \wedge$

$(\neg p@\{2..3\} \ \vee \ q.h \neq p.h \ \vee \ p.h < 0 \ \vee \ p.h \geq \lceil k/b \rceil)\} \ q.1$

$\{\neg p@\{2..3\} \ \vee \ q.h \neq p.h \ \vee \ p.h < 0 \ \vee \ p.h \geq \lceil k/b \rceil\}$, $q.h$ is not modified; also $q \neq p$.

$q@1 \ \wedge \ (\exists n : 0 \leq n < b :: \neg X[q.h][n]) \ \wedge \ p@\{2..3\} \ \wedge \ q.h = p.h \ \wedge$

$0 \leq p.h < \lceil k/b \rceil \wedge (\mathrm{A12})^{i,j}_{p.h,p.v}$

$\Rightarrow (\exists n : 0 \leq n < b :: \neg X[q.h][n]) \ \wedge \ p@\{2..3\} \ \wedge \ q.h = p.h \ \wedge$

$0 \leq p.h < \lceil k/b \rceil \ \wedge \ 0 \leq p.v < b \wedge (\mathrm{A12})^{i,j}_{p.h,p.v}$, by (I145).

$\Rightarrow (\exists n : 0 \leq n < b :: \neg X[q.h][n]) \ \wedge \ X[p.h, p.v] \ \wedge \ q.h = p.h$, by definition of (A12).

$\Rightarrow (\min n : 0 \leq n < b :: \neg X[q.h, n]) \neq p.v$, predicate calculus.

$\{q@1 \ \wedge \ (\exists n : 0 \leq n < b :: \neg X[q.h][n]) \ \wedge \ p@\{2..3\} \ \wedge \ q.h = p.h \ \wedge$

$0 \leq p.h < \lceil k/b \rceil \wedge (\mathrm{A12})^{i,j}_{p.h,p.v}\} \ q.1 \ \{q.v \neq p.v\}$

, by above derivation and program text.

For (A12), assume that $0 \leq i < \lceil k/b \rceil \ \wedge \ 0 \leq j < b$. (A12) can be falsified by statements that modify X, establish or falsify $r@\{2..3\}$, or modify $r.h$ or $r.v$ for some r. The statements to check are $r.0$, $r.1$, and $r.3$. Statement $r.0$ does not modify X; also $r@\{2, 3\}$

(and hence $r@\{2,3\} \ \wedge \ r.h = i \ \wedge \ r.v = j$) is false both before and after the execution of $r.0$. To show that $r.1$ does not falsify (A12), we consider the following four cases.

$\{r@1 \ \wedge \ (\forall n : 0 \leq n < b :: X[r.h][n]) \wedge (\text{A12})\}\ r.1\ \{r@1 \wedge (\text{A12})\}$

, by the program text, $r.1$ does not modify $X[i][j]$, and the loop does not terminate; also the pre- and post-conditions imply $\neg(r@\{2..3\} \ \wedge \ r.h = i \ \wedge \ r.v = j)$.

$\{r@1 \ \wedge \ (\exists n : 0 \leq n < b :: \neg X[r.h][n]) \ \wedge \ r.h \neq i \wedge (\text{A12})\}\ r.1\ \{r.h \neq i \wedge (\text{A12})\}$

, $r.1$ does not modify $X[i][j]$ because $r.h \neq i$;

also pre- and post-conditions imply $\neg(r@\{2..3\} \ \wedge \ r.h = i \ \wedge \ r.v = j)$.

$\{r@1 \ \wedge \ (\exists n : 0 \leq n < b :: \neg X[r.h][n]) \ \wedge \ r.h = i \ \wedge$

$(\min n : 0 \leq n < b :: \neg X[r.h][n]) = j \wedge (\text{A12})\}$

$r.1\ \{X[i][j] \ \wedge \ r@\{2,3\} \ \wedge \ r.h = i \ \wedge \ r.v = j\}$

, by program text.

$\{r@1 \ \wedge \ (\exists n : 0 \leq n < j :: \neg X[i][n]) \ \wedge \ r.h = i \ \wedge$

$(\min n : 0 \leq n < b :: \neg X[r.h][n]) \neq j \wedge (\text{A12})\}$

$r.1\ \{X[i][j] \equiv (\exists r :: r@\{2,3\} \ \wedge \ r.h = i \ \wedge \ r.v = j)\}$

, the precondition implies the postcondition; $r.1$ does not modify $X[i][j]$, establish $r@\{2,3\} \ \wedge \ r.h = i \ \wedge \ r.v = j$, or affect $q@\{2,3\} \ \wedge \ q.h = i \ \wedge \ q.v = j$ for $q \neq r$.

To show that $r.3$ does not falsify (A12), we consider the following two cases.

$\{r@3 \ \wedge \ (r.h \neq i \ \vee \ r.v \neq j) \wedge (\text{A12})\}\ r.3\ \{r@0 \ \wedge \ (r.h \neq i \ \vee \ r.v \neq j) \wedge (\text{A12})\}$

, $r.3$ does not modify $X[i][j]$, establish $r@\{2,3\} \land r.h = i \land r.v = j$, or affect $q@\{2,3\} \land q.h = i \land q.v = j$ for $q \neq r$.

$\{r@3 \land r.h = i \land r.v = j \land \text{(A11)}\}\ r.3$

$\{\neg X[i][j] \land r@0 \land (\forall s : s \neq r :: \neg s@\{2..3\} \lor s.h \neq i \lor s.v \neq j)\}$

, because $0 \leq i < \lceil k/b \rceil$, the precondition implies that $0 < r.h < \lceil k/b \rceil$; thus, by definition of (A11), the precondition implies $(\forall s : s \neq r :: \neg s@\{2..3\} \lor s.h \neq i \lor s.v \neq j)$, which is not falsified by $r.3$; also, $r.3$ establishes $\neg X[i][j] \land r@0$ in this case. □

The following invariant shows that, for each i, $0 \leq i < \lceil k/b \rceil$, there are always enough names left for the number of processes seeking names from $X[i]...X[\lceil k/b \rceil - 1]$.

invariant $0 \leq i < \lceil k/b \rceil \Rightarrow (|\{p :: p@\{1..3\} \land p.h \geq i\}| \leq k - ib)$ (I147)

Proof: By (I142), (I147) holds if $i = 0$. Henceforth, assume $0 < i < \lceil k/b \rceil$. Initially $(\forall p :: p@0)$ holds, and because $i < \lceil k/b \rceil$, it follows that $k - ib \geq 0$, so (I147) holds initially. (I147) can be falsified only by establishing $q@1$ or by incrementing $q.h$ for some process q. The statements to check are $q.0$ and $q.1$. After statement $q.0$ is executed, $q.h < i$ holds because $i > 0$. Statement $q.1$ can establish $q@\{1..3\} \land q.h \geq i$ only if executed when $q@1 \land q.h = i - 1$ holds. To show that $q.1$ does not falsify (I147) in this case, we consider the following two cases.

$\{q@1 \land q.h = i - 1 \land (\exists n : 0 \leq n < b :: \neg X[i-1][n]) \land \text{(I147)}\}\ q.1$

$\{q@2 \land q.h = i - 1 \land \text{(I147)}\}$

, by program text; loop terminates because $q.1$ establishes $q.v < b$.

$$q@1 \ \wedge \ q.h = i-1 \ \wedge \ (\forall n : 0 \le n < b :: X[i-1][n] \wedge (\text{I146})^{i,j}_{i-1,n}) \wedge (\text{I147})^{i}_{i-1}$$

$$\Rightarrow q@1 \ \wedge \ q.h = i-1 \ \wedge \ |\{p :: p@\{2..3\} \ \wedge \ p.h = i-1\}| \ge b \wedge (\text{I147})^{i}_{i-1}$$

, (I146) implies (A12); recall that $0 < i < \lceil k/b \rceil$.

$$\Rightarrow q@1 \ \wedge \ q.h = i-1 \ \wedge \ |\{p :: p@\{2..3\} \ \wedge \ p.h = i-1\}| \ge b \ \wedge$$
$$|\{p :: p@\{1..3\} \ \wedge \ p.h \ge i-1\}| \le k - ib + b$$

, definition of (I147).

$$\Rightarrow |\{p :: p@\{1..3\} \ \wedge \ p.h \ge i)\}| \le k - ib - 1$$

, predicate calculus; note that $q.h = i-1 \Rightarrow q.h \ge i-1 \ \wedge \ \neg(q.h \ge i)$.

$$\{q@1 \ \wedge \ q.h = i-1 \ \wedge \ (\forall n : 0 \le n < b :: X[i-1][n] \wedge (\text{I146})^{i,j}_{i-1,n}) \wedge (\text{I147})^{i}_{i-1}\} \ q.1 \ \{(\text{I147})\}$$

, by above derivation; $q.1$ does not establish $p@\{1..3\} \ \wedge \ p.h \ge i$ for $p \ne q$.$\square$

The following invariant shows that if a process reaches $X[\lceil k/b \rceil - 1]$, then its *set_first_zero* will succeed, so it will acquire a name.

invariant $p@1 \ \wedge \ p.h = \lceil k/b \rceil - 1 \Rightarrow (\exists n : 0 \le n < k - b(\lceil k/b \rceil - 1) :: \neg X[\lceil k/b \rceil - 1][n])$

(I148)

Proof: Consider the following derivation.

$$p@1 \ \wedge \ p.h = \lceil k/b \rceil - 1 \wedge (\text{I147})^{i}_{\lceil k/b \rceil - 1}$$

$$\Rightarrow p@1 \ \wedge \ p.h = \lceil k/b \rceil - 1 \ \wedge \ (|\{p :: p@\{1..3\} \ \wedge \ p.h \ge \lceil k/b \rceil - 1)\}| \ \le \ k - b(\lceil k/b \rceil - 1))$$

, by (I147).

$$\Rightarrow p.h = \lceil k/b \rceil - 1 \ \wedge \ (|\{p :: p@\{2..3\} \ \wedge \ p.h = \lceil k/b \rceil - 1\}| \ < \ k - b(\lceil k/b \rceil) - 1)$$

, predicate calculus.

$$\Rightarrow |\{n : 0 \le n < k - b(\lceil k/b \rceil - 1) :: X[\lceil k/b \rceil - 1][n]\}| \ < \ k - b(\lceil k/b \rceil - 1)$$

, observe that $0 \leq n < k - b(\lceil k/b \rceil - 1)$ implies $0 \leq n < b$; thus, by (I146),

$$|\{n : 0 \leq n < k - b(\lceil k/b \rceil - 1) :: X[\lceil k/b \rceil - 1][n]\}| \leq$$

$$|\{p :: p@\{2..3\} \wedge p.h = \lceil k/b \rceil - 1\}|.$$

$\Rightarrow(\exists n : 0 \leq n < k - b(\lceil k/b \rceil - 1) :: \neg X[\lceil k/b \rceil - 1][n])$, pigeonhole principle. □

The following invariants are used to show that process p acquires a name in $\{0, ..., k-1\}$ from one of the first $\lceil k/b \rceil$ segments of names.

invariant $p@1 \Rightarrow 0 \leq p.h < \lceil k/b \rceil$ (I149)

Proof: Initially $p@0$ holds, so (I149) holds. Only statements $p.0$ and $p.1$ affect (I149). Because $k > 1$ and $b > 0$, (I149) holds after $p.0$ is executed. Statement $p.1$ can falsify (I149) only if executed when $p.h = \lceil k/b \rceil - 1$. However, by (I148), $(\exists n : 0 \leq n < k - b(\lceil k/b \rceil - 1) :: \neg X[p.h][n])$ holds before $p.1$ is executed in this case. This implies that $(\exists n : 0 \leq n < b :: \neg X[p.h][n])$, so the antecedent does not hold after $p.1$ is executed. □

invariant $p@\{2,3\} \Rightarrow 0 \leq p.h < \lceil k/b \rceil - 1 \vee (p.h = \lceil k/b \rceil - 1 \wedge 0 \leq p.v < k - b(\lceil k/b \rceil - 1))$ (I150)

Proof: Initially, $p@0$ holds, so (I150) holds. Only statements $p.0$ and $p.1$ potentially falsify (I150). The antecedent does not hold after $p.0$ is executed. For $p.1$ we have the following.

$p@1 \wedge (p.h < 0 \vee p.h \geq \lceil k/b \rceil) \Rightarrow \mathit{false}$, by (I149).

$\{p@1 \wedge 0 \leq p.h < \lceil k/b \rceil - 1\}\ p.1\ \{p@1 \vee 0 \leq p.h < \lceil k/b \rceil - 1\}$

, if $p.1$ increases $p.h$ then it also assigns $v := b$, so the loop does not terminate.

$\{p@1 \wedge p.h = \lceil k/b \rceil - 1\}\ p.1\ \{p@2 \wedge p.h = \lceil k/b \rceil - 1 \wedge 0 \leq p.v < k - b(\lceil k/b \rceil - 1)\}$

, by (I148) and program text. □

Claim 21: Let c, d, c', and d' be nonnegative integers satisfying $(c \neq c' \ \vee \ d \neq d') \ \wedge \ 0 \leq d < b \ \wedge \ 0 \leq d' < b$. Then, $bc + d \neq bc' + d'$.

Proof: The claim is straightforward if $c = c'$, so assume that $c \neq c'$. Without loss of generality assume that $c < c'$. Then,

$$\begin{aligned} bc + d &< b(c+1) && , d < b. \\ &\leq bc' && , c < c'. \\ &\leq bc' + d' && , d' \geq 0. \end{aligned}$$ □

invariant $p \neq q \ \wedge \ p@3 \ \wedge \ q@3 \Rightarrow p.name \neq q.name$ (I151)

Proof: Consider the following derivation.

$p \neq q \ \wedge \ p@3 \ \wedge \ q@3$

$\Rightarrow p \neq q \ \wedge \ p@3 \ \wedge \ q@3 \ \wedge \ 0 \leq p.h < \lceil k/b \rceil$, by (I150).

$\Rightarrow p \neq q \ \wedge \ p@3 \ \wedge \ q@3 \ \wedge \ (q.h \neq p.h \ \vee \ q.v \neq p.v)$, by (A11) ((I146) implies (A11)).

$\Rightarrow (q.h \neq p.h \ \vee \ q.v \neq p.v) \ \wedge \ 0 \leq p.v < b \ \wedge \ 0 \leq q.v < b \ \wedge \ p.h \geq 0 \ \wedge \ q.h \geq 0$

, by (I144) and (I145).

$\Rightarrow b(p.h) + p.v \neq b(q.h) + q.v$, by Claim 21 with $c = p.h$, $d = p.v$, $c' = q.h$, and $d' = q.v$.

$\Rightarrow p.name \neq q.name$, by (I143). □

This concludes the proof that no two processes in their working sections have the same name. The following invariant shows that that each process acquires a name ranging over $0..k-1$.

invariant $p@3 \Rightarrow 0 \le p.name < k$ (I152)

Proof: Initially $p@0$ holds, so (I152) holds. Only statement $p.2$ potentially falsifies (I152). To show that $p.2$ does not falsify (I152), we consider the following three cases.

Case 1: $p@2 \ \wedge \ (p.h < 0 \ \vee \ p.h \ge \lceil k/b \rceil) \Rightarrow \mathit{false}$, by (I150).

Case 2: $p@2 \ \wedge \ 0 \le p.h < \lceil k/b \rceil - 1$

$\Rightarrow (0 \le b(p.h) \le k - b) \ \wedge \ (0 \le p.v < b)$, by (I145) and predicate calculus.

$\{p@2 \ \wedge \ 0 \le p.h < \lceil k/b \rceil - 1\} \ p.2 \ \{0 \le p.name < k\}$

, by the above derivation and the program text.

Case 3: $p@2 \ \wedge \ p.h = \lceil k/b \rceil - 1$

$\Rightarrow (p.h = \lceil k/b \rceil - 1) \ \wedge \ (0 \le p.v < k - b(\lceil k/b \rceil - 1))$, by (I150).

$\Rightarrow 0 \le (b(p.h) + p.v) < (b(\lceil k/b \rceil - 1) + k - b(\lceil k/b \rceil - 1))$

, predicate calculus, $b > 0$, $k > 0$.

$\Rightarrow 0 \le (b(p.h) + p.v) < k$, predicate calculus.

$\{p@2 \ \wedge \ p.h = \lceil k/b \rceil - 1\} \ p.2 \ \{0 \le p.name < k\}$

, by the above derivation and the program text. $\Box$

(I151) and (I152) prove that the algorithm shown in Figure 7.7 correctly implements long-lived k-renaming.

Observe that each time a shared variable is accessed when acquiring a name, either the loop terminates or $p.h$ is incremented. Thus, by (I149), p executes at most $\lceil k/b \rceil$ shared accesses before the loop terminates. Also, releasing a name requires 1 shared variable access. Thus, we have the following result.

Theorem 23 Using *set_first_zero* and *clr_bit* on b-bit variables, wait-free, long-lived k-renaming can be implemented so that the worst-case time complexity of acquiring and releasing a name once is $\lceil k/b \rceil + 1$. □

C.3 Correctness Proof for Algorithm in Figure 7.9

We inductively assume correctness for the right instance of $\lceil k/2 \rceil$-renaming and the left instance of $\lfloor k/2 \rfloor$-renaming. In accordance with the problem specification, we assume that the following invariant holds.

invariant $|\{p :: p@\{1..7\}\}| \leq k$ (I153)

The following two invariants follow directly from the program text in Figure 7.9.

invariant $p@\{5..6\} \Rightarrow p.side = right$ (I154)

invariant $p@7 \Rightarrow p.side \neq right$ (I155)

Proofs for the remaining invariants are provided below. Although each of the following two assertions is an invariant in its own right, it is convenient to prove that their conjunction is an invariant because this way we may inductively assume that both hold before any statement execution. These assertions are used to prove that too many processes do not access the left and right instances. This is required so that the correctness of these instances can be used to prove the algorithm correct inductively.

$0 \leq X \leq \lceil k/2 \rceil$ (A13)

$$|\{p :: p@2 \;\vee\; (p@\{4..7\} \;\wedge\; p.side = right)\}| = \lceil k/2 \rceil - X \tag{A14}$$

invariant (A13) $\wedge$ (A14) (I156)

Proof: Initially (A13) $\wedge$ (A14) holds. (A13) can only be falsified by decrementing X when $X = 0$ holds, or by incrementing X when $X = \lceil k/2 \rceil$ holds. By the definition of *bounded_decrement*, the first case does not arise. Only statement $p.6$ increments X. However, consider the following.

$p@6 \;\wedge\; X = \lceil k/2 \rceil \;\wedge\; p.side \neq right \wedge \text{(I154)} \Rightarrow false$, by (I154).

$p@6 \;\wedge\; X = \lceil k/2 \rceil \;\wedge\; p.side = right \wedge \text{(A14)} \Rightarrow false$, by definition of (A14).

(A14) is potentially falsified by any statement that modifies $p.side$ or X, or establishes or falsifies $p@2$ or $p@\{4..7\}$. The statements to check are $p.1$, $p.2$, $p.3$, $p.6$, and $p.7$ where p is any process. Statement $p.2$ preserves $p@2 \;\vee\; (p@\{4..7\} \;\wedge\; p.side = right)$ and statement $p.3$ preserves $\neg(p@2 \;\vee\; (p@\{4..7\} \;\wedge\; p.side = right))$. Also, neither statement modifies X. By (I154), statement $p.6$ decreases both sides of (A14) by 1. By (I155), statement $p.7$ does not affect either side. The following assertions imply that statement $p.1$ does not falsify (A14).

$p@1 \;\wedge\; X < 0 \wedge \text{(A13)} \Rightarrow false$, definition of (A13).

$\{p@1 \;\wedge\; X = 0 \wedge \text{(A14)}\}\; p.1 \;\{p@3 \wedge \text{(A14)}\}$

, by definition of *bounded_decrement*, $p.1$ does not modify X.

$\{p@1 \;\wedge\; X > 0 \wedge \text{(A14)}\}\; p.1 \;\{p@2 \wedge \text{(A14)}\}$

, both sides of (A14) are increased by 1 in this case. $\square$

invariant $|\{p :: p@2 \ \vee \ (p@\{4..7\} \ \wedge \ p.side = right)\}| \leq \lceil k/2 \rceil$ (I157)

Proof: (I157) follows directly from (I156). □

invariant $|\{p :: p@3 \ \vee \ (p@\{4..7\} \ \wedge \ p.side = left)\}| \leq \lfloor k/2 \rfloor$ (I158)

Proof: Initially, $(\forall p :: p@0)$ holds, so (I158) holds because $k > 0$. (I158) is potentially falsified by any statement that establishes $p@3 \ \vee \ (p@\{4..7\} \ \wedge \ p.side = left)$ for some p. The statements to check are $p.1$, $p.2$, and $p.3$. For statement $p.2$, we have $\{p@2\}\ p.2\ \{p@4 \ \wedge \ p.side = right\}$. Statement $p.3$ preserves $p@3 \ \vee \ (p@\{4..7\} \ \wedge \ p.side = left)$. The following assertions imply that statement $p.1$ does not falsify (I158).

$p@1 \ \wedge \ X < 0 \Rightarrow false$, by (A13) ((I156) implies (A13)).

$\{p@1 \ \wedge \ X > 0\}\ p.1\ \{p@2\}$, definition of *bounded_decrement*.

$p@1 \ \wedge \ X = 0$

$\Rightarrow p@1 \ \wedge \ |\{q :: q@2 \ \vee \ (q@\{4..7\} \ \wedge \ q.side = right)\}| = \lceil k/2 \rceil$

, by (A14) ((I156) implies (A14)).

$\Rightarrow |\{q :: q@3 \ \vee \ (q@\{4..7\} \ \wedge \ q.side = left)\}| < \lfloor k/2 \rfloor$, by (I153).

$\{p@1 \ \wedge \ X = 0\}\ p.1\ \{|\{q :: q@3 \ \vee \ (q@\{4..7\} \ \wedge \ q.side = left)\}| \leq \lfloor k/2 \rfloor$

, by preceding derivation; $p.1$ increases the left-hand side of (I158) by at most 1. □

By (I157) and (I158), the right instance is accessed by at most $\lceil k/2 \rceil$ processes concurrently and the left instance is accessed by at most $\lfloor k/2 \rfloor$ processes concurrently. By assumption, these instances are correct. Therefore, the following invariants follow easily from the correctness conditions.

invariant $p@\{4,5\} \ \wedge \ p.side = right \Rightarrow 0 \leq p.name < \lceil k/2 \rceil$ (I159)

invariant $p@\{4,7\} \ \wedge \ p.side = left \Rightarrow \lceil k/2 \rceil \leq p.name < k$ (I160)

invariant $p \neq q \ \wedge \ p@\{4..7\} \ \wedge \ q@\{4..7\} \ \wedge \ p.side = q.side \Rightarrow p.name \neq q.name$ (I161)

Correctness of the k-renaming algorithm shown in Figure 7.9 follows from (I159), (I160), and (I161). Note that, given the assumption that the left and right instances are correct, wait-freedom is trivial. This allows us to prove the following result.

Theorem 24 Using b-bit variables and *bounded_decrement* and *fetch_and_add*, wait-free, long-lived k-renaming can be implemented so that the worst-case time complexity of acquiring and releasing a name once is $2\lceil \log_2 k \rceil$ for $k \leq 2(2^b - 1)$.

Proof: By induction on k.

Basis: $k = 2$. 1-renaming can be trivially implemented with no shared accesses. Thus, in this case, the algorithm in Figure 7.9 implements 2-renaming with two shared accesses.

Induction: $k > 2$. Inductively assume that $\lceil k/2 \rceil$-renaming and $\lfloor k/2 \rfloor$-renaming can be implemented with time complexity at most $2\lceil \log_2 \lceil k/2 \rceil \rceil$ and $2\lceil \log_2 \lfloor k/2 \rfloor \rceil$, respectively. Thus, the algorithm in Figure 7.9 has time complexity at most $2 + 2\lceil \log_2 \lceil k/2 \rceil \rceil = 2 + 2\lceil \log_2 k - 1 \rceil = 2\lceil \log_2 k \rceil$, so the theorem holds. Note that because the shared counter X must be represented with b bits, this algorithm can only be implemented if $\lceil k/2 \rceil \leq 2^b - 1$. Thus, the proof only holds if $k \leq 2(2^b - 1)$. □

As noted in Section 7.4.2, the *set_first_zero* and *clr_bit* operations can be used to further improve the time complexity of this algorithm by "chopping off" the bottom $\lfloor \log_2 b \rfloor$

levels of the tree. This approach yields the following result.

Theorem 25: Using b-bit variables and *set_first_zero*, *clr_bit*, *bounded_decrement*, and *fetch_and_add*, wait-free, long-lived k-renaming can be implemented so that the worst-case time complexity of acquiring and releasing a name once is $2(\lceil \log_2 \lceil k/b \rceil \rceil + 1)$ for $1 \leq k \leq 2(2^b - 1)$.

Proof: By induction on k.

Basis: $k \leq b$. By Theorem 23, wait-free k-renaming can be implemented with time complexity $\lceil k/b \rceil + 1 = 2 = 2(\lceil \log_2 \lceil k/b \rceil \rceil + 1)$ when $k \leq b$.

Induction: $k > b$. Inductively assume that $\lceil k/2 \rceil$-renaming and $\lfloor k/2 \rfloor$-renaming can each be implemented with time complexity $2(\lceil \log_2 \lceil k/2b \rceil \rceil + 1) = 2\lceil \log_2 \lceil k/b \rceil \rceil$. Then, the algorithm in Figure 7.9 implements the k-renaming with time complexity at most $2 + 2\lceil \log_2 \lceil k/b \rceil \rceil = 2(\lceil \log_2 \lceil k/b \rceil \rceil + 1)$ shared accesses. As for Theorem 24, this proof only holds if $k \leq 2(2^b - 1)$.

$\square$

C.4 Correctness Proof for Algorithm in Figure 7.10

The differences between the safety proofs for the algorithms shown in Figures 7.9 and 7.10 are captured by the following three invariants. These invariants are easy to prove, and are therefore stated without proof.

invariant $|\{p :: (p@2 \ \wedge \ p.side = none) \ \vee \ (p@\{3..9\} \ \wedge \ p.side = right)\}| = \lceil k/2 \rceil - X$

(I162)

invariant $|\{p@\{3..9\} \;\wedge\; p.side = right\}| \leq \lceil k/2 \rceil$ (I163)

invariant $|\{p@\{3..9\} \;\wedge\; p.side = left\}| \leq \lfloor k/2 \rfloor$ (I164)

These invariants are analogous to (A14), (I157), and (I158), respectively. As with the proof for the algorithm shown in Figure 7.9, (I163) and (I164) are used to show that the left and right instances are not accessed by too many processes concurrently. The rest of the proof is similar to the previous one. The lock-freedom property for the algorithm shown in Figure 7.10 is captured formally by the following property.

Lock-Freedom: If a non-faulty process p attempts to reach its working section, then eventually some process (not necessarily p) reaches its working section.

Proof: We inductively assume that the left and right instances are lock-free. Thus, it is easy to see that the only risk to lock-freedom is that some non-faulty process p executes statements $p.1$ and $p.2$ forever, without any other process reaching its working section. Assume, towards a contradiction, that process p repeatedly executes statements $p.1$ and $p.2$. Consider consecutive statement executions, of $p.2$ and $p.1$, respectively. By the assumption that the loop executes repeatedly, it follows that $X > 0$ holds immediately after statement $p.2$ is executed, and that $X \leq 0$ holds immediately before statement $p.1$ is executed. Thus, X is decremented at least once between the execution of statements $p.2$ and $p.1$. Consider the first such decrement by some process q. The only statement that decrements X is statement $q.1$. As $q.1$ is the first decrement of X after the execution of $p.2$, it follows that $X > 0$ holds when $q.1$ is executed. Thus, $q.1$ establishes $q@3 \;\wedge\; q.side = right$. Note that process q can only decrement X again after reaching its working section. Thus, if some process p repeats the loop at $p.1$ and $p.2$ N times, then some process q reaches its working section. $\square$

Because a process may repeatedly execute statements $p.1$ and $p.2$ (while other processes make progress), the worst-case time complexity for the algorithm in Figure 7.10 is unbounded. However, if no other process takes a step between statements $p.1$ and $p.2$ being executed, then the test at statement $p.2$ will succeed. Therefore, if there is no contention, then the number of shared accesses generated by a process acquiring and releasing a name once is at most 2 plus the contention-free time complexity for the inductively-assumed instances. Thus, by an inductive proof similar to the proof of Theorem 24, we have the following result. This result can be extended, as Theorem 24 was in the previous section, to give a result analogous to Theorem 25.

Theorem 26: Using b-bit variables and *fetch_and_add*, lock-free, long-lived k-renaming can be implemented so that the worst-case, contention-free time complexity of acquiring and releasing a name once is $2\lceil \log_2 k \rceil$ for $k \leq 2(2^b - 1)$. $\square$

Bibliography

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic Snapshots of Shared Memory", *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, pp. 1-14.

[2] 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.

[3] A. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "A Bounded, First-In, First-Enabled Solution to the ℓ-Exclusion Problem", *ACM Transactions on Programming Languages and Systems*, 16(3), 1994, pp. 939-953.

[4] J. Anderson, "Composite Registers", *Distributed Computing*, 6(3), 1993, pp. 141-154.

[5] J. Anderson, "Multi-Writer Composite Registers", *Distributed Computing*, 7(4), 1994, pp. 175-195.

[6] J. Anderson and B. Grošelj, "Beyond Atomic Registers: Bounded Wait-Free Implementations of Nontrivial Objects", *Science of Computer Programming*, 1992, pp. 192-237.

[7] J. Anderson and M. Moir. "Towards A Necessary and Sufficient Condition for Wait-Free Synchronization", *Proceedings of the Seventh International Workshop on Distributed Algorithms*, 1993, pp. 39-53.

[8] J. Anderson and M. Moir, "Using k-Exclusion to Implement Resilient, Scalable Shared Objects", *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994, pp. 141-150.

[9] 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.

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

[11] R. Anderson and H. Woll, "Wait-Free Parallel Algorithms for the Union-Find Problem", *Proceedings of the 23rd ACM Symposium on Theory of Computing*, 1991, pp. 370-380.

[12] T. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, 1(1), 1990, pp. 6-16.

[13] J. Aspnes and M. Herlihy, "Wait-Free Data Structures in the Asynchronous PRAM Model", *Proceedings of the Second Annual ACM Symposium on Parallel Architectures and Algorithms*, 1990, pp. 340-349.

[14] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk, "Achievable Cases in an Asynchronous Environment", *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, 1987, pp. 337-346.

[15] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk, "Renaming in an Asynchronous Environment", *Journal of the ACM* 37(3), 1990, pp. 524-548.

[16] H. Attiya, M. Herlihy, and O. Rachman, "Efficient Atomic Snapshots Using Lattice Agreement", *Proceedings of the 6th International Workshop on Distributed Algorithms*, 1992, pp. 35-53.

[17] H. Attiya and O. Rachman, "Atomic Snapshots in $O(n \log n)$ Operations", *Proceedings of the 12th Annual ACM Symposium on the Principles of Distributed Computing*, 1993, pp. 29-40.

[18] 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.

[19] A. Bar-Noy and D. Dolev, "Shared Memory versus Message-Passing in an Asynchronous Distributed Environment", *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, 1989, pp. 307-318.

[20] BBN Advanced Computers, *Inside the TC2000 Computer*, February, 1990.

[21] B. Berhsad. "Practical Considerations for Non-Blocking Concurrent Objects", *Proceedings of the 13th International Conference on Distributed Computing Systems*, 1993, pp. 264-274.

[22] B. Bloom, "Constructing Two-Writer Atomic Registers", *IEEE Transactions on Computers*, 37(12), December 1988, pp. 1506-1514. Also appeared in *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 249-259.

[23] E. Borowsky and E. Gafni, "Immediate Atomic Snapshots and Fast Renaming", *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, 1993, pp. 41-50.

[24] H. Buhrman, J. Garay, J. Hoepman, and M. Moir, "Long-Lived Renaming Made Fast", *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995, pp. 194-203.

[25] J. Burns and G. Peterson, "Constructing Multi-Reader Atomic Values from Non-Atomic Values", *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 222-231.

[26] J. Burns and G. Peterson, "The Ambiguity of Choosing", *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, 1989, pp. 145-157.

[27] K. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[28] R. Cypher, "The Communication Requirements of Mutual Exclusion", *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1995, pp. 147-156.

[29] E. Dijkstra, "Solution to a Problem in Concurrent Programming Control", *Communications of the ACM*, 8(9), 1965, p. 569.

[30] D. Dolev, E. Gafni, and N. Shavit, "Towards a Non-atomic Era: l-Exclusion as a Test Case", *Proceedings of the 20th ACM Symposium on Theory of Computing*, 1988, pp. 78-92.

[31] D. Dolev and N. Shavit, "Bounded Concurrent Timestamp Systems are Constructible!", *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, 1989, pp. 454-465.

[32] C. Dwork, M. Herlihy, S. Plotkin, and O. Waarts, "Time Lapse Snapshots", *Proceedings of the Israel Symposium on the Theory of Computing and Systems*, 1992, pp. 154-170.

[33] C. Dwork and O. Waarts, "Simple and Efficient Bounded and Concurrent Timestamping or Bounded Concurrent Timestamp Systems are Comprehensible!", *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, 1992, pp. 655-666.

[34] M. Fischer, N. Lynch, J. Burns, and A. Borodin, "Resource Allocation with Immunity to Process Failure", *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, 1979, pp. 234-254.

[35] M. Fischer, N. Lynch, J. Burns, and A. Borodin, "Distributed FIFO Allocation of Identical Resources Using Small Shared Space", *ACM Transactions on Programming Languages and Systems*, 11(1), 1989, pp. 90-114.

[36] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process", *Journal of the ACM*, 1985, pp. 374-382.

[37] R. Gawlick, N. Lynch, and N. Shavit, "Concurrent Timestamping Made Simple", *Proceedings of the Israel Symposium on the Theory of Computing and Systems*, 1992, pp. 171-183.

[38] G. Graunke and S. Thakkar, "Synchronization Algorithms for Shared-Memory Multiprocessors", *IEEE Computer* 23, 1990, pp. 60-69.

[39] S. Haldar and P. Subramanian, "Space-Optimum Conflict-Free Construction of 1-Writer 1-Reader Multivalued Atomic Variable", *Proceedings of the 8th International Workshop on Distributed Algorithms*, 1994, pp. 116-128.

[40] S. Haldar and K. Vidyasankar, "Space-Efficient Construction of Buffer-Optimal 1-Writer 1-Reader Multivalued Atomic Variable", *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, 1996, p. 178.

[41] M. Herlihy, "Impossibility and Universality Results for Wait-Free Synchronization", *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, 1988, pp. 276-290.

[42] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Structures", *Proceedings of the Second Annual ACM Symposium on Principles and Practice of Parallel Programming*, 1990, pp. 197-206.

[43] M. Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems*, 13(1), 1991, pp. 124-149.

[44] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", *ACM Transactions on Programming Languages and Systems*, 15(5), 1993, pp. 745-770.

[45] M. Herlihy and J. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures", *Proceedings of the 20th International Symposium in Computer Architecture*, 1993, pp. 289-300.

[46] M. Herlihy and N. Shavit, "The Asynchronous Computability Theorem for t-Resilient Tasks", *Proceedings of the 25th ACM Symposium on Theory of Computing*, 1993, pp. 111-120.

[47] M. Herlihy and J. Wing, "Axioms for Concurrent Objects", *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pp. 13-26, 1987.

[48] 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.

[49] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM* 12, 1969, pp. 576-580,583.

[50] J.-H. Hoepman and J. Tromp, "Binary Snapshots", *Proceedings of the Seventh International Workshop on Distributed Algorithms*, 1993, pp. 18-25.

[51] M. Inoue, W. Chen, T. Masuzawa, and N. Tokura, "Linear-Time Snapshot Using Multi-Writer Multi-Reader Registers", *Proceedings of the 8th International Workshop on Distributed Algorithms*, 1994, pp. 130-140.

[52] A. Israeli and M. Li, "Bounded Time-Stamps", *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 1987, pp. 371-382.

[53] A. Israeli and L. Rappoport, "Efficient Wait-Free Implementation of a Concurrent Priority Queue", *Proceedings of the 7th International Workshop on Distributed Algorithms*, 1993, pp. 1-16.

[54] 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*, 1994, pp. 151-160.

[55] A. Israeli, A. Shaham, and A. Shirazi, "Linear-Time Snapshots for Unbalanced Systems", *Proceedings of the Seventh International Workshop on Distributed Algorithms*, 1993, pp. 26-38.

[56] P. Jayanti and S. Toueg, "Some Results on the Impossibility, Universality, and Decidability of Consensus", *Proceedings of the 6th International Workshop on Distributed Algorithms*, 1992, pp. 69-84.

[57] T. Johnson and K. Harathi, "Interruptible Critical Sections", Technical Report, TR94-007, Dept. Of Computer Science, University of Florida at Gainesville, 1993.

[58] L. Kirousis, E. Kranakis, and P. Vitanyi, "Atomic Multireader Register", *Proceedings of the Second International Workshop on Distributed Computing*, 1987, pp. 278-296.

[59] L. Kirousis, P. Spirakis, and P. Tsigas, "Reading Many Variables in One Atomic Operation: Solutions with Linear or Sublinear Complexity", *Proceedings of the Fifth International Workshop on Distributed Algorithms*, 1991, pp. 229-241.

[60] D. Knuth, "Additional Comments on a Problem in Concurrent Programming Control", *Communications of the ACM*, 9(5) 1966, pp. 321-322.

[61] A. LaMarca, "A Performance Evaluation of Lock-Free Synchronization Protocols", *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994, pp. 130-140.

[62] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem", *Communications of the ACM*, 17(8), 1974, pp. 453-455.

[63] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", *IEEE Transactions on Computers*, C-28(9), 1979, pp. 690-691.

[64] L. Lamport, "Specifying Concurrent Program Modules", *ACM Transactions on Programming Languages and Systems*, 5(2), 1983, pp. 190-222.

[65] L. Lamport, "A Fast Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems*, 5(1), 1987, pp. 1-11.

[66] L. Lamport, "On Interprocess Communication, Parts I and II", *Distributed Computing* 1, 1986, pp. 77-101.

[67] V. Lanin and D. Shasha, "Concurrent Set Manipulation without Locking", *Proceedings of the 7th Annual ACM Symposium on Principles of Database Systems*, 1988, pp. 211-220.

[68] M. Li, J. Tromp, and P. Vitanyi, "How to Construct Wait-Free Variables", *Proceedings of International Colloquium on Automata, Languages, and Programming*, 1989, pp. 488-505.

[69] B.-H. Lim and A. Agrawal, "Waiting Algorithms for Synchronization", *ACM Transactions on Computer Systems*, 11(3), 1993, pp. 253-294.

[70] M. Loui, H. Abu-Amara, "Memory Requirements for Agreement Among Unreliable Asynchronous Processes", *Advances in Computing Research* 4, 1987, pp. 163-183.

[71] H. Massalin and C. Pu, "A Lock-Free Multiprocessor OS Kernel", Technical Report CUCS-005-91, Columbia University, 1991.

[72] J. Mellor-Crummey and M. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, 9(1), 1991, pp. 21-65.

[73] M. Michael and M. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms", *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, 1996, pp. 267-276.

[74] M. Moir and J. Anderson, "Fast, Long-Lived Renaming", *Proceedings of the 8th International Workshop on Distributed Algorithms*, 1994, pp. 141-155.

[75] M. Moir and J. Anderson, "Wait-Free Algorithms for Fast, Long-Lived Renaming", *Science of Computer Programming* 25, 1995, pp. 1-39.

[76] M. Moir and J. Garay, "Fast, Long-Lived Renaming Improved and Simplified", to appear in the proceedings of the 10th International Workshop on Distributed Algorithms, 1996. A brief announcement appeared in *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, 1996, p. 152.

[77] S. Plotkin, "Sticky Bits and Universality of Consensus", *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, 1989, pp. 159-175.

[78] R. Newman-Wolfe, "A Protocol for Wait-Free, Atomic, Multi-Reader Shared Variables", *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 232-248.

[79] G. Peterson, "Concurrent Reading While Writing", *ACM Transactions on Programming Languages and Systems* 5, 1983, pp. 46-55.

[80] G. Peterson, personal communication, November 1995.

[81] G. Peterson and J. Burns, "Concurrent Reading While Writing II: The Multi-Writer Case", *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, 1987.

[82] G. Peterson and M. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System", *Proceedings of the 9th ACM Symposium on Theory of Computing*, 1977, pp. 91-97.

[83] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time System Synchronization", *IEEE Transactions on Computers*, 39(9), 1990, pp. 1175-1185.

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

[85] A. Singh, J. Anderson, and M. Gouda, "The Elusive Atomic Register", *Journal of the ACM*, 41(2), 1994, pp. 311-339.

[86] J. Singh, W. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory", *ACM SIGARCH Computer Architecture News*, 22(4), 1992, pp. 5-44.

[87] E. Styer, "Improving Fast Mutual Exclusion", *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, 1992, pp. 159-168.

[88] J. Tromp, "How to Construct an Atomic Variable", *Proceedings of the Third International Workshop on Distributed Algorithms*, 1989, pp. 292-302.

[89] J. Turek, D. Shasha, and S. Prakash, "Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Non-Blocking", *Proceedings of the 11th Symposium on Principles of Database Systems*, 1992, pp. 212-222.

[90] J. Valois, "Implementing Lock-Free Queues", *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, 1994, pp. 64-69.

[91] J. Valois, "Lock-Free Linked Lists Using Compare-and-Swap", *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995, pp. 214-222.

[92] J. Valois, "*Lock-Free Data Structures*", Ph.D. Thesis, Rensselaer Polytechnic Institute, May 1995.

[93] P. Vitanyi and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware", *Proceedings of the 27th IEEE Symposium on the Foundations of Computer Science*, 1986, pp. 233-243.

[94] J. Wing and C. Gong, "A Library of Concurrent Objects and their Proofs of Correctness", Technical Report CMU-CS-90-151, Carnegie Mellon University, 1990.

[95] J. Wing and C. Gong, "Testing and Verifying Concurrent Objects", *Journal of Parallel and Distributed Computing*, 17(2), 1993, pp. 164-182.

[96] R. Wisniewski, L. Kontothanassis, and M. Scott, "Scalable Spin Locks for Multiprogrammed Systems", *Proceedings of the 8th International Parallel Processing Symposium*, 1994, pp. 583-589.

[97] J.-H. Yang and J. Anderson, "Fast, Scalable Synchronization with Minimal Hardware Support", *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, 1993, pp. 171-182.

[98] J.-H. Yang and J. Anderson, "A Fast, Scalable Mutual Exclusion Algorithm", *Distributed Computing* 9, 1995, pp. 51-60.